VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Systém pro spouštění a monitorování výpočtu neuronových sítí

System for Running and Monotoring Machine Learning Algorithms

Bc. Tomáš Kovačik

Diplomová práce

Vedoucí práce: Ing. David Ježek, Ph.D.

Ostrava, 2021

Abstrakt

Tato diplomová práce se zabývá vývojem systému, který by umožňoval monitorování učení neuronových sítí a spouštění jejich výpočtů formou úloh v prostředí vzdáleného výpočetního serveru a superpočítače. V práci je uveden popis zaměřený na monitorování učení neuronových sítí, řízení úloh ve vzdálených prostředích a na možnosti perzistence metrik. Součástí práce je i popis implementace tří hlavních částí vyvíjeného systému, které zahrnují řídící server, klientskou knihovnu a uživatelské rozhraní.

Klíčová slova

neuronové sítě, monitorování, Java, superpočítač, webová aplikace

Abstract

This diploma thesis is focused on the development of a system that would allow monitoring of neural network learning and starting their calculations in the environment of a remote computational server and supercomputer. As part of this work is description of neural network learning monitoring, task management in remote environments and methods of metric persistence. This thesis also describes implementation of three main parts, which include management server, client library and user interface.

Keywords

neural network, monitoring, Java, supercomputer, web application

Poděkování

Rád bych na tomto místě poděkoval svému vedoucímu práce Ing. Davidu Ježkovi, Ph.D. za vedení a věcné připomínky k vypracování této diplomové práce.

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
CPU	–	Central Processing Unit
FTP	–	File Transfer Protocol
GPU	–	Graphics Processing Unit
HPC	–	High Performance Computing
HTTP	–	Hypertext Transfer Protocol
HTTPS	–	Hypertext Transfer Protocol Secure
IO	–	Input Output
JPA	–	Java Persistence API
ORM	–	Object Relational Mapping
PID	–	Process Identifier
REST	–	Representational State Transfer
SQL	–	Structured Query Language
SSD	–	Solid State Drive
SSE	–	Server Sent Events
SSH	–	Secure Shell
URL	–	Uniform Resource Locator

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Učení neuronových sítí představuje náročný proces, jehož výsledek je přímo závislý na mnoha faktorech. Mezi ně bychom mohli zařadit například použitý algoritmus, trénovací data, parametry neuronové sítě nebo počet iterací. S učením neuronových sítí proto často souvisí i provádění experimentů jejichž cílem je najít optimální nastavení tak, aby naučená síť splňovala požadavky, které na ni klademe.

Cílem této práce je vyvinout systém, který by umožňoval monitorování výpočtů spojených s učením neuronových sítí a jejich spouštění ve formě úloh na vzdáleném výpočetním serveru nebo superpočítači. Součástí práce by měla být i klientská knihovna s jejíž pomocí by bylo možné u jakékoliv úlohy zaznamenat potřebné metriky a přehledné uživatelské rozhraní pomocí něhož by bylo možné úlohy spravovat a naměřené metriky vizualizovat.

První část práce se věnuje problematice monitorování učení neuronových sítí, spouštění jejich výpočtů ve vzdálených prostředích a perzistenci metrik. Jsou zde rozebrány vybrané přístupy k záznamu metrik, jejich výhody a nevýhody a v neposlední řadě také možnosti dvou specializovaných databází pro jejich ukládání. Současně jsou zde představeny i dva základní přístupy ke spouštění a řízení úloh spolu s popisem prostředí superpočítače a jeho porovnáním s obecným výpočetním serverem.

Druhá část je věnována implementaci řídícího serveru, kde je detailněji rozebrána architektura projektu, použité schéma databáze a implementace několika důležitých komponent a služeb. Velký důraz je zde kladen na implementaci částí souvisejících se spouštěním úloh a jejich řízením ve vzdálených prostředích.

Ve třetí a čtvrté části je popsána implementace klientské knihovny a uživatelského rozhraní. U knihovny je představeno její poskytované rozhraní spolu s implementací základních a rozšiřujících funkcí. V případě uživatelského rozhraní jsou následně popsány vybrané sekce týkající se například správy úloh nebo vizualizace metrik.

Poslední dvě části obsahují popis nasazení aplikace do produkčního prostředí a souhrn dosažených výsledků spolu s uvedenými náměty pro budoucí vylepšení.

Kapitola 2

Analýza problému

Tato kapitola se věnuje problematice monitorování učení neuronových sítí a možnostem spouštění úloh, které učení neuronových sítí realizují.

2.1 Problematika monitorování učení neuronových sítí

Monitorování učení neuronových sítí představuje jednu z velice důležitých úloh, která nám umožňuje zásadním způsobem ovlivnit nejenom kvalitu výsledného řešení, ale i celkovou časovou náročnost. S její aplikací se lze především setkat při vývoji modelů neuronový sítí, kde s pomocí trasování a záznamu metrik můžeme cíleně upravovat nejrůznější parametry sítě, tak abychom docílili co největší přesnosti, účinnosti a efektivity výsledného modelu a procesu učení.

Proces monitorování má významný přínos ale i při vývoji samotných algoritmů neuronových sítí, kde bez pomoci sledování chování učení bývá často ladění a hledání případných chyb problematickou záležitostí. Za tento důsledek může převážně rostoucí složitost a rozsáhlost neuronových sítí u kterých nelze jednoduše pouhým pohledem ať už na kód algoritmu nebo stav proměnných určit, zdali se učení chová podle očekávání. Díky možnostem trasování nejrůznějších parametrů a metrik můžeme ale takovéto chyby snáze odhalit včas již při vývoji.

Metrik, které můžeme tímto způsobem získat je celá řada a častokrát se liší u jednotlivých typů neuronových sítí. Mezi ty pravděpodobně nejčastěji sledované bychom mohli zařadit například:

- Hodnoty chyb
- Hodnoty vah
- Pravděpodobnost aktivace skrytých neuronů

Existuje ale i řada metrik, které lze sledovat bez ohledu na typ učení a mezi něž patří údaje o využívání výpočetních prostředků jako:

- Využití CPU
- Využití operační paměti
- Využití GPU
- Aktivita IO

2.1.1 Způsoby záznamu metrik

K sledování a zaznamenání metrik lze využít řadu řešení, jejichž výběr je obvykle závislý na plánovaném způsobu použití, vizualizace, způsobu uložení a sdílení a v neposlední řadě i na prostředí, ve kterém se chystáme učení provádět. V následujících částech jsou detailněji popsány, a nakonec porovnány nejčastěji používané přístupy.

2.1.2 Záznam metrik do konzole

Za základní a zároveň nejjednodušší řešení můžeme považovat manuální záznam metrik s výstupem do konzole bez dalšího zpracování. Je zřejmé, že nevýhody budou v tomto případě převažovat, jelikož takto získané údaje budou nepřehledné, orientace v nich bude velice složitá a vizualizace téměř nemožná. Také nám tento typ přístupu v budoucnu neumožní srovnání mezi jednotlivými experimenty vlivem chybějícího verzování nebo sdílení mezi vícero lidmi. Z těchto důvodů je toto řešení spíše vhodné při počátečních pracích během vývoje, kde se obvykle pracuje s jednoduššími typy neuronových sítí nebo při prototypování.

2.1.3 Záznam metrik do interní datové struktury nebo souboru

Dalším řešením může být použití záznamu metrik do interní datové struktury nebo souboru, tak aby bylo možné je v rámci stejné aplikace později načíst a vizualizovat. Tento způsob je již o něco praktičtější, jelikož nám dává značnou volnost ve volbě formátu výstupních dat a výsledné grafické reprezentace, která tak často může být přizpůsobena zcela na míru našim požadavkům. Za velkou výhodu lze zde taky považovat možnost volby mezi vykreslením grafického výstupu do okna aplikace nebo do souboru, díky čemuž je možné aplikaci využívat i v rámci prostředí, které nedisponují grafickým uživatelským rozhraním. S těmito možnostmi je toto řešení uplatnitelné téměř kdekoliv, avšak vzhledem k tomu, že vyžaduje implementaci vlastních komponent, je obvykle jeho využití časově náročnější.

2.1.4 Záznam metrik do externího systému

Posledním řešením, se kterým se běžně můžeme setkat využívá záznam metrik do externí aplikace. Ta následně zajišťuje jejich zpracování a verzování, tak aby bylo možné každý experiment porovnat a vyhodnotit. Součástí těchto aplikací obvykle bývá široká podpora vizualizace v podobě grafů nebo

možnosti správy konfigurací, které jsou s daným experimentem svázány. Pro případ kolaborativního přístupu, kdy na vývoji neuronové sítě nebo jejího modelu spolupracuje tým lidí, je běžně součástí těchto řešení i přístup přes webové rozhraní. Díky tomu je možné výsledky včetně vizualizací hned sdílet mezi kolegy nebo k nim přistupovat téměř odkudkoliv. Stejně jako u ostatních řešení, i zde lze najít několik nevýhod v podobě vyšších nároků na komunikaci a na infrastrukturu v případě kdy je daný systém provozován v interní síti. V současné době je k dispozici celá řada platforem a aplikací realizující zmíněné funkce, které se liší svým rozsahem, možnostmi, cílovými uživateli a cenou.

2.1.5 Volba řešení způsobu záznamu metrik

S přihlédnutím k zamýšlenému způsobu využití v rámci implementace vlastního systému pro správu a spouštění úloh dlouhodobých výpočtů neuronových sítí, se jeví jako nejvhodnější řešení záznamu metrik do externího systému. V porovnání s ostatními způsoby nabízí centralizovaný přístup ke sběru a zpracování metrik, možnost okamžitého náhledu na získané data a snadnou integraci s komponentami umožňující řízení a správu experimentů. Vzhledem k tomuto výběru bude nutné ale vyřešit problémy spojené s přenosem metrik prostřednictvím počítačové sítě a s perzistencí těchto metrik. Těmto tématům se detailněji věnuje kapitola 2.4 a dále pak kapitoly týkající se implementace.

2.2 Spouštění a řízení výpočetních úloh

Učení neuronových sítí s rostoucím objemem trénovacích dat a zvyšující se složitostí představují velmi výpočetně náročnou úlohu vyžadující výkonný hardware. I přestože se hranice výkonu běžných počítačů každým rokem posouvá značným způsobem dopředu, stále nemusí být jejich výkon dostatečný na to, aby se na nich učení neuronové sítě uskutečnilo v rozumném čase.

Běžně se tak můžeme setkat s využitím specializovaných výpočetních clusterů nebo superpočítačů, které disponují odpovídajícím hardwarem s podstatně vyšším výkonem.

Tyto výpočetní clustery jsou následně dostupné skrze zabezpečený protokol SSH umožňující připojení ke vzdálenému terminálu, přičemž jejich uživatelé se musí orientovat v prostředí serverového operačního systému a v jeho softwarovém vybavení.

V rámci vyvíjeného systému je proto implementován modul umožňující vzdálené spouštění a správu úloh učení neuronových sítí napsaných v programovacím jazyce Java ve dvou prostředích. Prvním z nich je prostředí obecného výpočetního uzlu, u kterého se předpokládá využití operačního systému Ubuntu. Druhé prostředí je pak představováno platformou HPC.

U vyvíjeného modulu byly zváženy celkem dvě řešení využívající jak již zmíněný přímý přístup skrze SSH protokol, tak i přístup pomocí speciálně navrženého klienta, který by běžel přímo na vzdáleném clusteru.

2.2.1 Spouštění úloh za využití klienta jako zprostředkovatele

Přístup využívající klienta jako zprostředkovatele představuje jednu z alternativních možností k připojení skrze SSH protokol, která nevyžaduje aktivní připojení ke clusteru. Toho je možné docílit pomocí aplikace nezávisle běžící na vzdáleném clusteru, která přijímá požadavky na spuštění nových úloh z řídícího serveru. Pro příjem nových požadavků je proto nutností, aby klientská aplikace byla přístupná z venkovní sítě a aktivně naslouchala novým spojením. Ke komunikace mezi řídícím serverem a klientskou aplikací by tak bylo možné využít například protokol HTTPS, který poskytuje šifrované spojení a umožňuje autentifikaci druhé strany.

Další podstatnou vlastností tohoto přístupu je vynechání nutnosti přihlášení k serveru za pomoci uživatelského jména a hesla při spouštění nebo zastavování úloh. Díky tomu lze poskytnout přístup k výpočetním prostředkům bez nutnosti zakládání dodatečných uživatelských účtů na serveru, což na druhou stranu spolu nese bezpečnostní riziko.

Praktickému využití ve vyvíjeném systému ale brání zejména omezení kladená nastavením síťové komunikace v prostředí HPC, kde není přístup z venkovní sítě skrze port 443 umožněn právě z důvodu zajištění bezpečnosti.

2.2.2 Spouštění úloh za využití protokolu SSH

Protokol SSH [1] představuje velmi výkonný mechanismus pro vzdálený přístup k síťovým zařízením, mezi nimiž můžeme najít i běžné počítače nebo servery. V rámci tohoto přístupu umožňuje spouštění příkazů na vzdáleném terminálu, přenos souborů mezi oběma stranami nebo tunelování připojení. Mezi jeho nejsilnější stránky ale patří především možnosti zabezpečení spojení, jimiž se liší oproti alternativním protokolům jako je Telnet nebo FTP. Spojení v tomto případě využívá architekturu klient-server a jeho prvotní vytvoření probíhá ve čtyřech krocích:

1. Kontaktování serveru klientem
2. Zaslání veřejného klíče serveru klientovi
3. Vzájemné vyjednání parametrů zabezpečeného spojení a otevření zabezpečeného kanálu
4. Přihlášení uživatele ke vzdálenému systému

Počáteční spojení se serverem je zahájeno klientem, který od serveru obdrží jeho veřejný klíč. Ten je následně porovnán klientem za využití kryptografie založené na veřejných klíčích s jeho lokální databází důvěryhodných hostů (obvykle reprezentované souborem *known_hosts*). Je-li nalezena shoda, klient následně přistoupí k vzájemnému vyjednání parametrů zabezpečeného spojení. V tomto kroku si server a klient vymění seznam podporovaných šifrovacích algoritmů a na základě nejlepší shody s ohledem na co nejvyšší zabezpečení je vybrán vhodný kandidát. Následně je vytvořen šifrovaný kanál, který je využit během další komunikace zahrnující autentifikaci klienta. Pro její

účely existuje několik variant zahrnujících klasický způsob přihlášení pomocí dvojice uživatelského jména a hesla nebo přihlášení pomocí tzv. SSH klíče. [2]

Základní variantu v tomto ohledu představuje autentifikace uživatele za pomoci jeho jména a hesla, které uživatel musí zadat při každém přihlášení. Nevýhoda této metody spočívá v nutnosti vytvoření dostatečně složitého hesla, jehož síla významně ovlivňuje bezpečnost celého procesu přihlášení.

Alternativu nevyžadující heslo v tomto ohledu poskytuje druhý ze způsobů autentifikace využívající SSH klíče. Tato varianta je založená na asymetrické kryptografii, jenž pro ověření využívá páru klíčů – veřejný a privátní. Veřejný klíč bývá přiřazen na serveru ke konkrétnímu uživateli nebo skupině uživatelů a není potřeba ho chránit před zneužitím. Kdokoliv vlastnící veřejný klíč ho může využít pro šifrování dat, které mohou být opětovně dešifrovány a přečteny pouze pomocí odpovídajícího privátního klíče. Z tohoto důvodu musí zůstat privátní klíč utajen na zařízení uživatele a nesmí být dále šířen. Vlastnictví privátního klíče tak prokazuje identitu uživatele a je možné ho využít při přihlášení místo hesla. Vzhledem ke kryptografické složitosti klíčů se jedná o mnohem bezpečnější způsob přihlášení než za pomoci pouhého hesla, které by se mu svou sílou nemohlo rovnat.

Pro vyšší bezpečnost lze i samotný privátní SSH klíč zašifrovat, a to pomocí klíče derivovaného z tzv. přístupové fráze (*passphrase*), tak aby se zabránilo případnému zneužití v případě, že se klíč dostane k někomu nepovolanému. Derivace samotného klíče je v tomto případě zajištěna pomocí hashovací funkce, která je aplikována na přístupovou frázi. [3]

Dokončením autentifikace uživatele je celý proces zahájení komunikace skrze SSH protokol u konce a vytvořené spojení je možné využít pro spouštění příkazů nebo přenos dat. K realizaci těchto funkcí můžeme využít některou z dostupných implementací určenou pro náš operační systém. U linuxových operačních systémů je jednou z takových implementací i klient OpenSSH. Jeho funkcí můžeme využívat prostřednictvím příkazu `ssh`, přičemž pro jednoduché přihlášení k serveru si vystačíme s příkazem majícím následující podobu: [4]

```
ssh hostname
```

Po spuštění tohoto příkazu budeme v případě našeho prvního připojení k danému serveru vyzváni k ověření otisku veřejného klíče, a to jeho ručním porovnáním. Potvrzení je dále následováno výzvou k zadání přihlašovacích údajů po jejichž úspěšné kontrole získáme přístup ke vzdálenému terminálu.

Pro spuštění pouze jednoho příkazu si můžeme vystačit i s druhou podporovanou variantou mající podobu:

```
ssh hostname command
```

Ta po přihlášení provede příkaz `command` a ukončí spojení se serverem.

Utilita ssh toho ale umí daleko více. Za zmínku určitě stojí i možnost přesměrování komunikace na určitém portu, kterou je možné realizovat lokálně nebo vzdáleně. U lokálního typu dochází k přesměrování komunikace z určitého portu klienta na zvolený port na serveru, zatímco u vzdáleného

typu je to naopak. Možnost lokálního přesměrování portů je tak užitečná zejména v případě kdy bychom chtěli přistupovat například k databázovému serveru, jehož port není vystaven do sítě, ve které se nacházíme. Vzdálené přesměrování portů můžeme naopak využít při potřebě komunikace některé z aplikací, která běží na serveru s například webovým serverem běžícím na našem zařízení.

Pro povolení lokálního směrování je potřeba přidat parametr specifikující port klientského zařízení jehož komunikace bude přesměrována a adresu s portem na kterou se bude směrovat:

```
ssh -L [bind_address]port:host:host_port hostname
```

Alternativně lze uvést i adresu, s kterou bude v případě klienta komunikace svázaná. U vzdáleného směrování vypadá příkaz obdobně, ale místo parametru `-L` je použit parametr `-R` a význam jednotlivých částí je pozměněn. Část označená jako port u vzdáleného typu představuje port na serveru, z něhož má být komunikace směrována na port lokálního zařízení, který je uveden v části `host_port` s adresou danou řetězcem host. Opět je také možnost specifikovat adresu na serveru, se kterou bude komunikace svázaná.

2.2.3 Volba řešení pro vzdálené spouštění úloh

Z předchozího popisu je patrné, že oproti způsobu spouštění úloh za využití klienta jako zprostředkovatele, je protokol SSH daleko vhodnější kandidát. Přispívá k tomu zejména svými možnostmi zabezpečení a poskytovanou funkcionalitou v podobě přímého spouštění příkazů, přenosu souborů nebo směrováním portů. Při výběru byly ale zohledněny možnosti samotného prostředí HPC, které počítá především s přístupem skrze SSH protokol a jehož nastavení síťové komunikace by neumožňovalo nasazení prvního z jmenovaných přístupů. Popis konkrétního způsobu využití protokolu SSH při implementaci je možné dále nalézt v části popisující praktickou realizaci aplikace jenž je zároveň hlavním předmětem této práce.

2.3 Prostředí spouštění úloh

V průběhu vývoje a učení neuronových sítí se můžeme mimo lokální prostředí běžně setkat s prostředím obecného výpočetního uzlu nebo superpočítače. Mezi jejich hlavní rozdíly bývá často zařazována hardwarová výbava, která do jisté míry bývá záměrně upravena pro určitý způsob využití. Může se tak jednat například o cluster osazený velmi vysokým počtem procesorových jader nebo akceleračními kartami, které dokážou významně urychlit výpočty spojené s učením neuronových sítí.

Se způsobem využití a hardwarem rovněž souvisí i výbava softwarová ovlivňující nejen efektivitu využívání jednotlivých zdrojů, ale také celkové možnosti.

2.3.1 Prostředí superpočítače

S využitím superpočítače se běžně můžeme setkat v řadě odvětví využívajících jeho masivní výkon pro nejrůznější výpočty a simulace.

K dosažení takto vysokého výkonu je obvykle využita početná skupina uzlů tvořena vlastními výpočetními jednotkami, které jsou mezi sebou propojeny vysokorychlostní sítí. Přiřazení těchto uzlů konkrétním úlohám pak probíhá za využití speciálního plánovače úloh, jehož hlavním cílem je zajistit co nejefektivnější využití všech výpočetních prostředků. Pro jeho dosažení je obvykle využita fronta s prioritami, ve které jsou postupně řazeny úlohy dle řady kritérií, mezi něž patří samotné nároky na výpočetní zdroje, požadovaný výpočetní čas, přístup k speciálním akcelerátorům nebo množství již využitého času. Konkrétní nastavení preferencí, jimiž jsou úlohy z fronty vybírány ale vždy závisí na konkrétním plánovači úloh a jeho nastavení.

Ten je v případě poskytnutého superpočítače Barbora zastoupen aplikací PBS Pro, na nějž cílí i aplikace vyvíjená v rámci této práce. Barbora nabízí pro alokaci zdrojů několik front specifických svými hardwarovými prostředky a nastavením. V jejich základním výčtu můžeme najít fronty jako například: [5]

qexp – expresní fronta určená pro testování a spouštění velmi krátkých úloh. Výpočetní čas je zde omezen pouze na 1 h a ke spouštění úloh není potřeba specifikace projektu. Maximálně je možné alokovat 8 uzlů na jednoho uživatele.

qprod – produkční fronta určená pro spouštění běžných úloh. Výpočetní čas je zde omezen na 48 h a ke spuštění úloh je potřeba specifikovat projekt s nenulovým zbytkovým stavem hodin. Pro běh úloh je k dispozici až 187 uzlů, kde každý z nich disponuje 36 jádry.

qlong – dlouhá produkční fronta určená pro spouštění dlouhotrvajících úloh. Maximální délka alokace výpočetního času je zde omezena na 144 h. Stejně jako u předchozí fronty je vyžadován projekt s dostatečným zůstatkem hodin. Pro běh úloh je k dispozici 20 uzlů s 36 jádry.

qnvidia, qfat – dedikované fronty, které umožňují využití akcelerovaných uzlů kartami NVIDIA. V rámci jednoho uzlu jsou k dispozici 4 tyto karty. Zařazení úloh do této fronty vyžaduje speciální pověření.

qfree – fronta umožňující využití volných prostředků v případě vyčerpání všech přiřazených zdrojů. Výpočetní čas je u této fronty omezen pouze na 12 h, přičemž je zde k dispozici 189 uzlů po 16 jádrech.

Žádost o alokaci konkrétních výpočetních prostředků je možné provést pomocí příkazu `qsub`. Tomu je možné předat několik parametrů zahrnujících cestu ke skriptu úlohy, specifikaci fronty, požadovaného času, počtu uzlů a v případě fronty jiné než *qexp* i identifikátoru projektu. Podoba obecného příkazu se zmíněnými parametry je vidět na následujícím řádku:

```
qsub -A Project_ID -q queue -l select=x:ncpus=y,walltime=[[hh:]mm:]ss[.ms] \
jobscript
```

Každá takto vytvořená žádost je následně zpracovávána plánovačem úloh, který zvolí optimální pořadí pro alokaci požadovaných výpočetních prostředků. Po jejich přidělení dochází ke spuštění skriptu reprezentující danou úlohu, a to na prvním alokovaném uzlu. Všechny přidělené prostředky jsou následně dostupné úloze pouze během omezeného času, po jehož uplynutí je úloha zastavena. [6]

Pro práci s daty má každá úloha k dispozici dva sdílené souborové systémy, které se liší svým účelem, výkonem a kapacitou: [7]

HOME – Souborový systém uchovávající všechna data nacházející se v uživatelských adresářích */home/[username]* u nichž má každý uživatel k dispozici 24 GB prostoru. Jeho použití cílí především na přípravu dat, jejich vyhodnocení a pozdější zpracování. Maximální propustnost činí 1 GB/s.

SCRATCH – Vysoce výkonný souborový systém určený pro dočasné uložení dat, které jsou generovány v průběhu výpočtů. Jeho použití také cílí na úlohy s vysokými nároky na rychlost přístupu k úložišti a jeho propustnost, jenž v tomto případě činí až 38 GB/s. Každý uživatel má k dispozici 9,3 TB.

Oba souborové systémy jsou dostupné ve všech výpočetních a řídicích uzlech.

Všechny uzly mají rovněž k dispozici i vysokorychlostní síť určenou pro jejich vzájemnou komunikaci. Ta je v prostředí superpočítače omezena pouze na interní síť a není tak možné mimo ni přímo navazovat spojení z výpočetních uzlů. Obdobná omezeni platí i pro uzly sloužící pro přihlášení a obsluhu, které mohou přímo komunikovat pouze skrze protokoly SSH, HTTP, HTTPS a RSYNC využívající porty 22, 80, 443 a 873.

Pro zajištění spojení za využití jiných portů a protokolů lze využít funkci pro přesměrování portů, kterou poskytuje protokol SSH. Této funkce lze rovněž využít i u výpočetních uzlů a pomocí dvou tunelů (z výpočetního uzlu na přihlašovací uzel a z něj poté na jiné zařízení) umožnit spojení s libovolným serverem nebo zařízením. Princip směrování portů a jeho nastavení je podrobněji popsán v kapitole 2.2.2.

2.3.2 Porovnání prostředí superpočítače a prostředí obecného výpočetního uzlu

Z pohledu vyvíjené aplikace je za prostředí obecného výpočetního uzlu považován běžný server, který disponuje serverovým operačním systémem a požadovaným softwarovým vybavením potřebným pro spouštění aplikací realizujících učení neuronových sítí.

Jeho zásadním rozdílem ve srovnání s prostředím superpočítače je především hardwarová konfigurace a s tím i spojený výpočetní výkon. Ten je řádově nižší a díky přítomnosti pouze jediného

uzlu není na něm obvykle přítomen ani specializovaný plánovač úloh, který by se staral o přidělování rozsáhlých zdrojů mezi úlohami. To má také za následek i rozdílný způsob spouštění úloh u kterých již není nutné specifikovat parametry prostředí uzlů, na kterých by se uskutečnil jejich běh. Z tohoto důvodu bude muset být přizpůsobena vyvíjená aplikace tak, aby zajistila pro každou úlohu její správnou parametrizaci, spuštění a následné řízení jejího životního cyklu.

Další změnu můžeme najít i v komunikaci, která v případě obecného výpočetního uzlu nemusí být tak striktně omezena jako u superpočítače. V tomto případě se proto není nutné omezovat při přenosu metrik nebo jiných zpráv pouze na protokoly HTTP nebo HTTPS, ale je možné využít i jiné specializované protokoly.

2.4 Perzistence metrik

Metriky svou povahou představují časově založené údaje, které jsou získávány pravidelným opakovaným měřením v průběhu určitého časového intervalu. Typicky se tak jedná o data získané z měření nejrůznějších senzorů, stavu a vytížení jednotlivých počítačových komponent například na serveru nebo o data získané měřením medicínských přístrojů.

Oproti běžným datům relační databáze se tento typ záznamů liší především v přítomnosti atributu nesoucí časový údaj, který hraje klíčovou roli při porovnání a analýze hodnot z časového hlediska a v přístupu k vkládání a změnám v datech. Právě v tomto přístupu dochází k nejvýraznějším rozdílům, neboť u záznamu metrik obvykle platí, že změny jejich hodnot jsou reprezentovány novými záznamy. Velmi zřídka nebo skoro vůbec tak nedochází k aktualizaci hodnot u starších záznamů, což má z následek nárůst objemu dat.

Volba vhodného úložiště je proto klíčová s ohledem na výkonnost a propustnost při vkládání nových záznamů a při následném dotazování. Pro jejich uložení tak existuje několik specializovaných databází orientovaných právě na práci s tímto typem dat. Dvěma z nich, kterými jsou databáze InfluxDB a TimescaleDB jsou dále věnovány samostatné části v této kapitole.

Ještě před jejich popisem budou ale představeny dva datové modely, které lze využít při ukládání a práce s metrikami. Těmi jsou datový model Narrow-table a Wide-table.

2.4.1 Datový model Narrow-table

Za jeden ze základních datových modelů používaných ve spojitosti s metrikami je považován model Narrow-table [8], který je charakteristický několika svými vlastnostmi:

- Každá metrika je reprezentována jako samostatná entita
- Každý záznam obsahuje dva atributy reprezentující čas a hodnotu
- Hodnoty metadat jsou představovány množinou tagů, které jsou spojeny se samotnou metrikou

timestamp	sensor_id	location_id	name	value
2020-04-16 01:00:00	s123	175	hudmity	57,6
2020-04-16 01:01:00	s123	175	pressure	1020
2020-04-16 01:02:00	s123	175	temperature	26,1

Tabulka 2.1: Ukázka modelu Narrow-table

Tento model tak zajišťuje to, že je každá kombinace metrik nebo množiny tagů považována za jednu časovou řadu obsahující jejich hodnoty. Jeho nevýhodou jsou ale vyšší nároky na kapacitu úložiště a rychlost zápisu nových záznamů do databáze.

Nejlepší uplatnění proto nalezne všude tam, kde jsou hodnoty metrik získávány nezávisle na ostatních a v případech, kdy využijeme možnosti dynamicky přidávat nové typy metrik nebo nové tagy. Praktickou ukázku použití modelu Narrow-table je možné vidět na tabulce 2.1.

2.4.2 Datový model Wide-table

U datového modelu Wide-table [8] jsou hodnoty metriky zaznamenány společně pod jeden časový údaj, díky čemuž může být zachován vztah mezi jednotlivými údaji.

Oproti datovému modelu Narrow-table přináší zejména výkonnostní zlepšení při ukládání, kdy je vytvořen pouze jeden záznam obsahující čas a hodnoty. To spolu nese i další výhody v podobě nižších nároků na úložiště a vyššího výkonu při dotazování, neboť při práci s vícero metrikami není potřeba záznamy spojovat pomocí konstrukce JOIN.

Z důvodu pevně daných atributů tabulek používajících tento model ale není možné dynamicky přidávat nebo jakkoliv upravovat metriky, které mohou být v tabulce uloženy, protože jsou reprezentovány právě množinou jejich atributů.

Podoba modelu Wide-table je vidět na tabulce 2.2.

2.4.3 TimescaleDB

TimescaleDB je vysoce optimalizovaná databáze pro práci s časově orientovanými daty implementovaná jako rozšíření relační databáze PostgreSQL. Tímto rozšířením je umožněno využití všech stávajících výhod a funkcí relační databáze mezi něž patří zejména možnost definovat vztahy a omezení,

timestamp	device_id	hudmity	pressure	temp	location_id
2020-04-16 01:00:00	s123	57,6	1020	26,1	175
2020-04-16 01:00:00	s456	65,4	1024	27,4	175
2020-04-16 01:00:00	s789	78,7	1018	25,2	175

Tabulka 2.2: Ukázka modelu Wide-table

a to i pro tabulky uchovávající metriky nebo možnost využití plné podpory dotazovacího jazyka SQL. Ten je v tomto případě rozšířen o řadu nových specializovaných funkcí zaměřených na práci s časově orientovanými daty, které umožňují jejich pokročilou analýzu a zpracování. [9]

Mezi výhody této kombinace lze zařadit i možnost využití všech existujících nástrojů a knihoven třetích stran kompatibilních se samotným PostgreSQL. Díky tomu je její použití a integrace velice snadná a obvykle nevyžaduje žádné dodatečné úpravy nebo znalost dalšího dotazovacího jazyka.

Ve srovnání s databází PostgreSQL přináší v oblasti časově orientovaných dat podstatné výhody, a to zejména:

- Schopnost rychlého ukládání dat i u objemné databáze
- Vysoký výkon při dotazování
- Specializované funkce zaměřené na práci s časovými daty

K dosažení co nejvyššího výkonu byla zvolena vlastní architektura pro ukládání záznamů jejímž základem je datová struktura nazývaná jako *Hypertable*, která poskytuje abstrakci nad řadou několika individuálních tabulek nesoucí data nazývaných jako *Chunks*. [10]

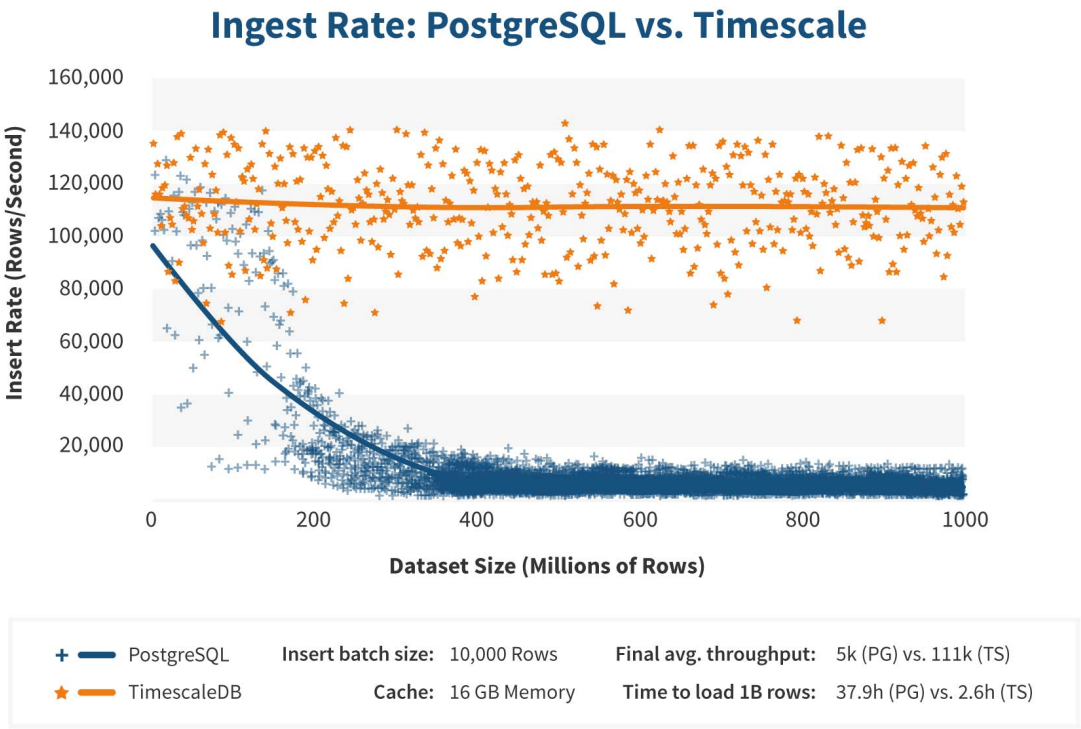
Každá dílčí tabulka je v rámci tabulky typu *Hypertable* vytvořena dělením všech dat do jedné nebo více dimenzí, přičemž základní rozdělení je prováděno podle časového intervalu. Dodatečně pak mohou být data rozdělena podle dalších klíčů jako například uživatelského identifikátoru, umístění senzoru, identifikátoru zařízení nebo kteréhokoliv jiného atributu.

Dělení na malé části pak umožňuje zajistit to, aby všechny B-stromy pro jednotlivé indexy tabulek mohly být během vkládání nových záznamů umístěny v operační paměti. Díky tomu lze podstatně snížit dopady na výkon způsobené přesunem jednotlivých částí velkého B-stromu mezi operační pamětí a diskem při aktualizaci jeho náhodných částí.

Velikost dílčích tabulek (*chunks*) proto hraje důležitou roli při vkládání, kde dosažením toho, že se poslední dílčí tabulka s nejnovějšími daty a její B-strom vleze do operační paměti, můžeme podstatně ovlivnit celkový výkon.

To se ostatně projevilo i v experimentu demonstrující rozdíl výkonu při ukládání metrik v porovnání s databází PostgreSQL, který provedli experti z TimescaleDB. Během něj se pokusili nasimulovat běžný scénář počítající s ukládáním dat generovaných monitoringem, ve kterém postupně po dávkách vložili jednu miliardu záznamů. Na testované instanci Azure VM (DS4 v2, 8 jader, SSD úložiště), tak dosáhli výsledné propustnosti přibližně 111 000 záznamů/s, která je mnohem vyšší v porovnání s propustností 5 000 záznamů/s u databáze PostgreSQL. Výsledek měření v podobě grafu je možné vidět na obrázku 2.1. [11]

Kromě vyššího výkonu nabízí TimescaleDB i specializované funkce v oblasti práce s časově orientovanými daty, které tradiční relační databáze nepodporují. Mezi nimi bychom mohli najít například funkce:



Obrázek 2.1: Porovnání výkonu databáze TimescaleDB s PostgreSQL [11]

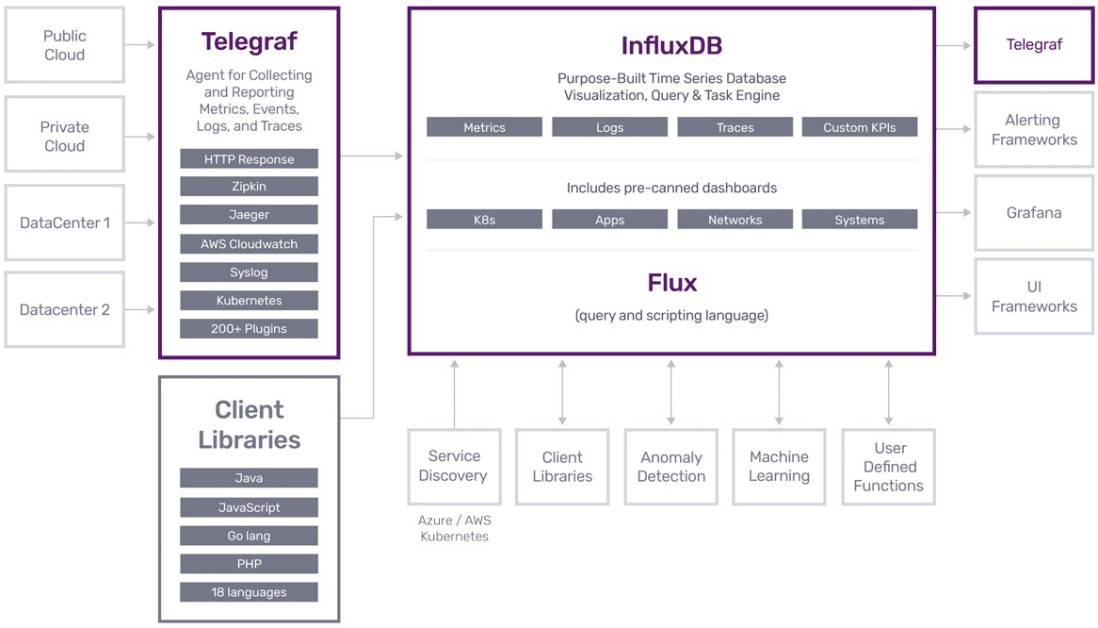
time_bucket – poskytuje seskupování hodnot času do přesně definovaných intervalů. Oproti funkci `date_trunc`, která je součástí PostgreSQL umožňuje specifikovat interval s jemnější granularitou jako například interval *'5 minute'* [12]

first, last – umožňují získání první nebo poslední hodnoty určitého sloupce agregovaného podle sloupce jiného (musí obsahovat čas). Například pro získání první naměřené teploty vzhledem k časové posloupnosti by volání funkce mělo následující podobu: `first(temperature, time)` [12]

2.4.4 InfluxDB

InfluxDB je další z databází zaměřená na práci s časově orientovanými daty, která díky její možnosti zapojení do ekosystému platformy Infuxdata umožňuje její široké využití. V rámci této platformy jsou nabízeny nástroje pro sběr a reportování metrik, nástroje pro vizualizaci dat nebo například klientské knihovny. Možnosti jejich spojení jsou vidět na obrázku 2.2 [13].

Oproti ostatním databázím je InfluxDB specifická především svým návrhem úložiště, které využívá vlastní pevně definovaný model [14]. Ten se nejvíce podobá modelu *Narrow-table* a skládá se z částí:



Obrázek 2.2: Ekosystém platformy Influxdata [13]

fieldSet – obsahuje skupinu párů klíč/hodnota nesoucí data o metrikách, které se váží ke společnému časovému údaji *timestamp*

tagSet – obsahuje skupinu párů klíč/hodnota nesoucí metadata spojené s časovým údajem *timestamp* a polem *fieldSet*. Jedná se o nepovinný údaj.

measurement – sdružuje záznamy do jednoho logického celku. Jedná se v podstatě o ekvivalent tabulky.

timestamp – představuje čas samotného měření

Tento model zavádí i striktnější přístup k indexování, který je možný pouze u položek množiny tagů, nebo omezené množství použitelných datových typů. Ty umožňují u tagů využití pouze datového typu `String`, přičemž jejich hodnoty nelze u záznamů aktualizovat. Omezena je i možnost aktualizace atributu *fieldSet*, u které je možné pouze přidávat další položky, nikoliv je odebírat. Pro kompletní změnu záznamů je tak potřeba je odebrat a přidat znova.

Pro analýzu dat využívající daný model nabízí InfluxDB možnost využití dvou dotazovacích jazyků Flux a InfluxQL, které se liší nejen svými možnostmi ale i syntaxí. U jazyka InfluxQL se tak můžeme setkat se syntaxí velice podobnou jazyku SQL, avšak ve srovnání s ním nenabízí plnou podporu všech klauzulí a je uzpůsobený zejména pro práci s vlastním specifickým schématem pro uložení dat. Oproti němu Flux nese rysy funkcionálního programovacího jazyka a podporuje všechny funkce, které samotná databáze nabízí.

Mezi nimi bychom mohli nalézt i funkce specializované na práci s časově orientovanými daty, které z velké části nabízejí shodnou funkcionalitu jako u databáze TimescaleDB. Nechybí zde proto funkce `first` a `last` pro získání první nebo poslední hodnoty dle času nebo `histogram` umožňující získání dat pro vykreslení histogramu. [15]

2.4.5 Volba databázového systému

Při výběru databázového systému byly kladeny nejvyšší nároky na celkovou jednoduchost použití, na možnosti ukládání metrik a výkon při jejich vkládání a dotazování.

V rámci cílového použití ve vyvíjené aplikaci byla proto zvolena databáze TimescaleDB, která oproti databázi InfluxDB nabízí například snadnou integraci s relační databází s jejímž využitím se počítalo i při uchování dalších dat. Toto spojení s sebou přináší i možnosti využití důvěrně známého dotazovacího jazyka SQL obohaceného o rozšiřující funkce pro práci s časově orientovanými daty. Navíc se zde nabízí i možnost využití široké škály datových typů nebo definice integritních omezení.

Z pohledu propustnosti při vkládání nových záznamů a výkonu při jejich následném získání, poskytuje databáze TimescaleDB velmi dobré výsledky, které zaručují nekompromisní výkon i při náročnějším využití.

Kapitola 3

Aplikace řídícího serveru

Tato kapitola se zabývá implementací aplikace řídícího serveru. Je zde podrobně popsána architektura projektu, databázový model a implementace několika nejdůležitějších částí spojených například se spouštěním a řízením výpočetních úloh nebo analýzou metrik.

3.1 Použité technologie a knihovny

Pro vývoj aplikace řídícího serveru a klientské knihovny byla zvolena jako hlavní platforma Java, která představuje moderní a osvědčené řešení pro vývoj multiplatformních aplikací. Její uplatnění můžeme nalézt při tvorbě jak mobilních nebo desktopových aplikací, tak i u rozsáhlých podnikových systémů.

Mezi důvody, které stály za zvolením této platformy patří především umožnění vývoje aplikací nezávislých na platformě. Díky tomu je tak možné, aby klientská knihovna mohla být spuštěna téměř v jakémkoliv prostředí nezávisle na počítačové architektuře nebo operačním systému. K těmto prostředím bychom mohli zařadit i prostředí superpočítače nebo obecného výpočetního uzlu na které se zároveň soustředí tato práce. Dalším důvodem byla i snaha zajistit snadnou integraci do již existujícího projektu Modeleru neuronových sítí, v němž by měla knihovna pomoci při vývoji a ladění nových modulů.

Kromě samotné platformy bylo zvoleno i několik knihoven, aplikačních rámců a technologií. Výčet těch nejdůležitějších spolu s jejich stručným popisem je uveden níže.

Spring Boot – nástavba, jejímž cílem je usnadnit tvorbu samostatných aplikací a služeb založených na aplikačním rámci Spring. K tomu využívá zejména své možnosti podpory automatické konfigurace a široké nabídce rozšiřujících modulů, které zahrnují například balíčky pro práci s perzistentní vrstvou využívají JPA nebo balíčky poskytující zabezpečení. [16]

TimescaleDB – specializovaná databáze optimalizovaná pro práci s časově orientovanými daty implementovaná jako rozšíření relační databáze PostgreSQL. Její podrobnější popis je uveden v části 2.4.3. [10]

Hibernate – ORM rámec, který poskytuje implementaci specifikace JPA (Java Persistence API). V rámci ní umožňuje využití standardizovaného přístupu pro definici objektově-relačního mapování, které významně usnadňuje práci a komunikaci s databází na perzistentní vrstvě. Kromě něj dále umožňuje využití dotazovacího jazyka JPQL, který umožňuje tvorbu dotazů s využitím objektového přístupu. [17]

JOOQ – vysoce výkonná knihovna umožňující dynamickou a typově bezpečnou konstrukci SQL dotazů. Ta je založena na doménově specifickém jazyku, který poskytuje přehledný a čitelný zápis zejména u komplikovanějších dotazů. V rámci vyvíjené aplikace je využívaná při dynamickém sestavování dotazů pro získání dat metrik, jejichž kritéria jsou určené velkým množstvím atributů. [18]

Apache MINA SSHD – knihovna umožňující komunikaci se vzdálenými zařízeními pomocí protokolu SSH, která podporuje práci jak v režimu klienta, tak i serveru. Její součástí je podpora pro rozšiřující moduly, které umožňují využití například SFTP subsystému, SCP příkazu nebo modulu pro práci s PUTTY klíči. Ve vyvíjené aplikaci je využita zejména při komunikaci se superpočítačem a obecným výpočetním uzlem. [19]

3.2 Architektura projektu

Na nejvyšší úrovni je projekt složen z celkem tří částí zahrnujících frontend, backend a klientskou knihovnu, kde aplikace řídícího serveru je představována právě částí backend. Ta je dále dělena na dva moduly, které jsou vytvořené pomocí nástroje Maven a jenž představují její služby.

První z těchto modulů je *auth-service*, jehož hlavní účel spočívá v poskytnutí autentifikace a autorizace přístupu k ostatním částem systému. Tento modul zároveň spravuje uživatelské účty a jejich přístupové údaje a poskytuje své služby ostatním modulům pomocí předdefinovaného rozhraní.

K němu přistupuje další z modulů pojmenovaný jako *management-service*. Jeho rozsah služeb, které poskytuje zahrnuje kompletní správu zdrojů jako jsou uživatelské profily, projekty, výpočetní úlohy nebo další související entity. Pro účely manipulace s těmito zdroji je nabízeno rozhraní REST s příslušnými koncovými body definujícími možné operace. Toto rozhraní je následně využito i dvěma dalšími částmi zahrnující frontend a klientskou knihovnu. U části frontend jsou využívány jeho možnosti pro přehlednou správu všech zmíněných entit, zatímco klientská knihovna ho využívá pouze pro vkládání nových záznamů metrik, signalizaci stavu úlohy, nahrávání souborů nebo tvorbu tagů.

Modul *management-service* také v rámci poskytování svých služeb využívá i další externí systémy v podobě obecného výpočetního uzlu nebo superpočítače, které představují cílová prostředí pro

spouštění výpočetních úloh. Komunikace s nimi je zajištěna pomocí protokolu SSH, přičemž pro každé z prostředí jsou v aplikaci k dispozici komponenty s jednotným rozhraním umožňující práci v nich.

Na úrovni jednotlivých modulů je dále využita třívrstvá architektura jejíž vrstvy jsou v celkové struktuře odlišeny pomocí balíčků. Balíčky v projektu zároveň sjednocují i další logické celky, které se týkají dodatečné funkcionality zahrnující například komponenty pro správu a řízení konkrétních instancí jednotlivých výpočetních úloh.

3.3 Schéma databáze

Pro perzistenci dat v aplikaci byla zvolena databáze TimescaleDB, která je orientovaná na práci s časově založenými daty a jejíž základ tvoří standardní relační databáze PostgreSQL. V rámci ní byla vytvořena řada tabulek jejichž vazby a vlastnosti jsou zachyceny ve schématu, které bylo vytvořeno nástrojem Datagrip a jenž je vidět na obrázku 3.1.

Nejdůležitějším prvkem v tomto schématu je tabulka `projects`, která má za cíl uchovávat všechny informace o projektech. Ty v aplikaci vytvářejí prostor pro vývoj konkrétního modelu nebo skupiny souvisejících modelů neuronových sítí, do něhož spadají všechny informace okolo aplikačních souborů, datových sad, tagů, uživatelských profilů jednotlivých členů nebo výpočetních úloh.

K uchování těchto navázaných informaci slouží jejich odpovídající tabulky, které jsou s tabulkou projektů spojeny pomocí cizích klíčů odkazujících se na její primární klíč. Atributy těchto cizích klíčů jsou zároveň ve schématu označeny jako povinné, díky čemuž nemůže nastat situace, kde by existovaly dílčí záznamy například v tabulce `files` uchovávající metadata o projektových souborech bez toho, aniž by tyto záznamy byly s nějakým konkrétním projektem spojeny. Kromě zajištění povinnosti cizích klíčů je u všech vztahů vynucena také referenční integrita, díky níž se nemůže záznam například v tabulce `tasks` odkazovat na neexistující záznam v tabulce `projects`.

Podíváme-li se blíže na některé ze jmenovaných tabulek, můžeme u nich nalézt použití definice unikátního klíče, pomocí něhož je zajištěna unikátnost určitých hodnot napříč záznamy. U tabulky `project_tags` je tento mechanismus využit pro dvojici atributů `value` a `project_id`, díky čemuž nemůžou v této tabulce současně existovat jakékoliv dva záznamy mající stejnou kombinaci hodnot těchto atributů. Stejnou aplikaci takového klíče můžeme nalézt i v tabulce `project_assignments` nebo `files`, kde je taktéž zapotřebí zajistit unikátnost vybraných hodnot.

Kromě tabulky `projects`, můžeme ve schématu nalézt i velmi důležitou tabulku `tasks`, jejímž hlavním účelem je uchování konfiguračních nastavení pro jednotlivé výpočetní úlohy. Ty mohou být v aplikaci reprezentovány celkem dvěma typy, které umožňují definovat specifické nastavení týkající se jednotlivých prostředí, ve kterých mají být dané úlohy spuštěny.

Spojení konkrétních atributů s objektovými typy je v případě této tabulky zajištěno pomocí strategie *Single Table*, definující způsob mapování dědičnosti do databáze. Tato strategie zajišťuje

to, že každý z atributů podtypů i jejich společného nadtypu jsou mapovány do jediné tabulky, přičemž pro rozlišení konkrétního typu je použit jeden z atributů nadtypu. Ten je u tabulky `tasks` reprezentován rozlišujícím atributem `target_type`, přičemž v aplikaci je mu přiřazována jedna ze tří hodnot podle typu prostředí.

S využitím zmíněné strategie je spojena i jedna podstatná nevýhoda v podobě nemožnosti definice omezení *not-null* u typově specifických atributů. Jejich kontrola proto musí být provedena jinou cestou, například validací na aplikační úrovni, což je současně i varianta využitá v případě vyvíjené aplikace.

Ve spojení s definicemi výpočetních úloh se můžeme setkat i s další důležitou tabulkou, kterou je tabulka `metrics`. Ta je specifická především svým datovým modelem, jelikož se jedná o tzv. úzkou tabulku (*Narrow-table*). V ní je každá naměřená hodnota uložena jako jediný záznam s časovou značkou, hodnotou, tagem a případně s dalšími popisnými atributy. Díky tomu je možné do tabulky ukládat záznamy téměř jakýchkoliv metrik bez nutnosti redefinice jejího schématu.

Kromě datového modelu se tabulka s metrikami liší i svým typem, který je představován tzv. hyper-tabulkou (*hypertable*) používanou v rámci databáze TimescaleDB pro uchování časově orientovaných dat. Vytvoření této tabulky bylo provedeno za pomoci příkazu:

```
SELECT create_hypertable('metrics', 'record_time');
```

U něj si můžeme všimnout zejména hodnoty druhého argumentu, který určuje název sloupce s časovým údajem. Ten musí být zároveň součástí primárního klíče, jenž je v tomto případě tvořen atributy `record_time`, `tag_id` a `task_id`.

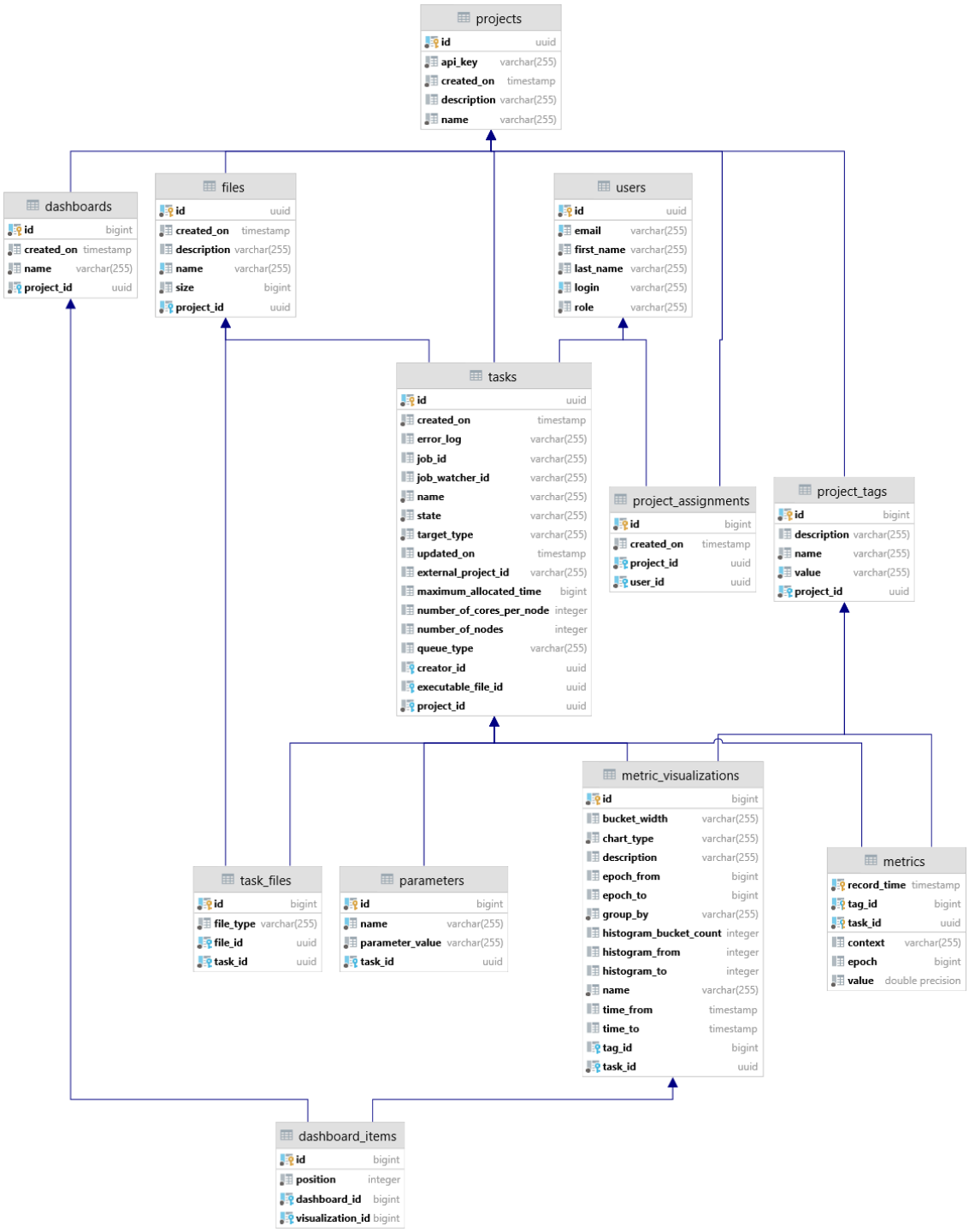
3.4 Správa datových souborů v projektu

Správa datových souborů potřebných ke spuštění a běhu výpočetních úloh je v aplikaci zajištěna pomocí služby `FileService`. Ta pro ukládání a načítání dat využívá kombinovaný přístup, v němž jsou binární data jednotlivých souborů uložena do souborového systému a jejich metadata do relační databáze. Metody implementující tyto základní operace proto využívají i další ze služeb pojmenovanou jako `StorageService`, která nabízí vysokoúrovňové rozhraní pro práci s externím úložištěm. S jejím využitím se tak můžeme setkat u následujících tří metod služby `FileService`, jejichž signatury jsou uvedeny ve výpisu 3.1.

```
UUID create(InputStream inputStream, GenericFile file);
void save(InputStream inputStream, GenericFile file);
Optional<FileResource> getFileResourceById(UUID id);
```

Výpis 3.1: Signatury metod rozhraní FileService

První z metod umožňuje nahrání nového souboru, jehož binární data jsou poskytnuta pomocí parametru `inputStream` a jeho metadata skrze parametr `file`. Po zavolání této metody jsou nejprve



Obrázek 3.1: Schéma databáze

uložena metadata do tabulky `files` a až poté jsou zapsána data z předaného datového proudu ve formě souboru na disk. K identifikaci tohoto souboru je pak jako název využit vygenerovaný

identifikátor získaný ze záznamu zapsaného do databáze. V případě, že záznam pro soubor s daným názvem již existuje, dojde k vyvolání výjimky a ukončení provádění dané metody.

Druhá z metod je téměř totožná s první až na to, že v případě existujícího záznamu daného souboru jsou jeho data přepsána těmi, které jsou poskytnuty pomocí datového proudu `inputStream`.

Poslední z uvedených metod umožňuje získání datového proudu a jeho metadat na základě předaného identifikátoru. V případě, že daný soubor není nalezen je vrácen prázdný objekt typu `Optional`.

3.5 Implementace funkcí pro analýzu metrik

Funkce pro analýzu metrik jsou v aplikaci zajišťovány pomocí tříd `MetricController`, `MetricService` a `MetricRepositoryExtension`, přičemž nejdůležitější roli hraje právě poslední z jmenovaných tříd. Ta má na starost interakci s databází pomocí dotazů, které jsou speciálně vytvořené z předaných parametrů a umožňuje získáni čtyř druhů výsledků. Signatury metod jsou vidět ve výpisu 3.2.

```
List<Plain2DPoint<Instant, Double>> findMetricsGroupedByTime(MetricCriteria
    metricCriteria, String bucketWidth);
List<Plain2DPoint<Long, Double>> findMetricsGroupedByEpoch(MetricCriteria
    metricCriteria);
List<Plain2DPoint<String, Double>> findMetricsGroupedByContext(MetricCriteria
    metricCriteria);
Integer[] findMetricsHistogram(MetricCriteria metricCriteria, double from, double
    to, int bucketCount);
```

Výpis 3.2: Signatury metod rozhraní MetricRepositoryExtension

Všechny uvedené metody využívají pro parametrizaci dotazů a následnou filtraci výsledků objekt typu `MetricCriteria`, jenž obsahuje povinné a nepovinné atributy vztahující se ke sloupcům tabulky `metrics`. Mezi těmi nepovinnými můžeme najít například atributy `timeFrom` a `timeTo` sloužící k výběru časového úseku nebo atributy `epochFrom` a `epochTo` určené k výběru rozsahu epochy. U povinných atributů se naopak můžeme setkat s atributem `taskId` a `tagId`, které určují v prvním případě identifikátor výpočetní úlohy, pro níž mají být výsledky získány a ve druhém případě identifikátor tagu, který zastupuje zvolenou metriku.

S uvedenými metodami jsou spojené i jejich návratové typy. Ty jsou ve většině případů tvořeny seznamem zastoupeným kolekcí typu `List`, jehož položky jsou datového typu `Plain2DPoint` představující dvojrozměrnou souřadnici bodu. Výjimku v tomto případě tvoří poslední metoda, která jako svůj návratový typ využívá pole typu `Integer`.

Přejdeme-li dále k implementaci jednotlivých metod, můžeme u každé z nich najít konstrukci odpovídajícího databázového dotazu, který realizuje její hlavní funkci. K tomu je v každé z me-

tod využita knihovna JOOQ, která prostřednictvím svého doménově specifického jazyka umožňuje dynamické sestavování SQL dotazů.

V případě první metody je její hlavní funkce tvořena dotazem vracející seznam hodnot metrik seskupených podle času. U něj byla využita speciální funkce `time_bucket` databáze TimescaleDB, která provádí zarovnání časového atributu do určitého úseku, jehož šířka je určena druhým parametrem s názvem `bucketWidth`. K agregaci výsledných hodnot byla nakonec použita funkce průměru.

U druhé a třetí z metod jsou jejich funkce dále zastoupeny dotazy umožňující získání seznamu metrik seskupených podle epochy a kontextu, přičemž obdobně jako u předchozí metody je pro agregaci výsledných hodnot opět použita funkce průměru.

Poslední z metod je nakonec zastoupena dotazem umožňujícím získat histogram hodnot podle zadaných kritérií. K tomu byla využita funkce `histogram`, která je rovněž implementována databází TimescaleDB. Ta pro sestavení svého výsledku využívá parametry metody `from`, `to` a `bucketCount`, které udávají spodní a vrchní hranici histogramu a počet skupin v něm.

3.6 Správa a řízení úloh

Výpočetní úlohy lze v kontextu aplikace chápat jako samostatně spustitelné jednotky, které jsou definovány svými vlastnostmi a konfigurací potřebnou pro jejich vykonání. Tato konfigurace je v aplikaci reprezentována množinou objektů entitních tříd, které slouží k její perzistenci a dalšímu zpracování. Za jednu ze základních entit je v tomto ohledu považována třída `TaskDO`, která zapouzdřuje obecné informace potřebné pro běh každé úlohy. Mezi jejími atributy můžeme najít například ty, které specifikují:

- Identifikátor spustitelného souboru
- Typ prostředí, ve kterém má být úloha spuštěna
- Parametry běhového prostředí

Identifikátor spustitelného souboru v tomto případě představuje pouze odkaz na daný soubor, který je reprezentován objektem třídy `FileDO`. Pro vytvoření úlohy musí být tedy samotný spustitelný soubor dopředu nahrán jako součást projektu, tak aby mohl být později použit pro její definici.

Typ prostředí u každé z úloh dále určuje, jakým způsobem má být inicializována a spuštěna, případně jaké komponenty mají být při její manipulaci použity. V rámci aplikace jsou podporovány celkem tři prostředí, mezi něž patří prostředí lokální, vzdálené a prostředí superpočítače.

Jejich nastavení je poté určeno pomocí atributů specifikujících jejich parametry. Ty se u každého z prostředí mohou lišit, a proto je třída `TaskDO` upravena tak, aby mohla být rozšiřována pomocí dědičnosti. V případě superpočítače, tak byla vytvořena její podtřída s názvem `HpcTaskDO`, která specifikuje několik dalších parametrů zahrnujících například:

- Typ fronty
- Počet uzlů
- Počet jader CPU na každém uzlu
- Maximální doba běhu úlohy
- ID projektu

Výše zmíněná základní konfigurace může být dále v případě potřeby rozšířena i dodatečnými parametry, které ovlivňují přímo úlohu. Mezi ně mohou patřit například:

- Identifikátory dodatečných souborů potřebných pro běh
- Parametry specifikující nastavení samotné úlohy

V případě dodatečných souborů je situace podobná jako u toho spustitelného, až na to, že tyto soubory jsou do cílového prostředí pouze nakopírovány, tak aby byly výpočetní úloze k dispozici. Současně také platí i pravidlo, že musí být tyto soubory předem nahrány jako součást projektu předtím, než budou použity.

Poslední část konfigurace se dále týká parametrizace samotné úlohy. Pro tu je možné specifikovat parametry ve formě klíč-hodnota, které budou předány aplikaci reprezentující danou úlohu ve formě argumentů příkazové řádky. Přístup k těmto parametrům je následně poskytován pomocí dodatečné klientské knihovny, která kromě funkcí monitoringu poskytuje právě i možnost práce s parametry.

S oběma rozšířeními je spojena i jejich definice, která je v aplikaci provedena pomocí entit `TaskFileDO` a `ParameterDO`. Ty jsou v aplikaci spojeny s konkrétní výpočetní úlohou, pro jejíž běh jsou následně využívány.

3.6.1 Vytváření a modifikace úloh

Pro vytváření a modifikaci samotných úloh jsou aplikací vystavovány koncové body rozhraní REST, jejichž mapování na metody je prováděno ve třídě `TaskController`. Ta pro tyto operace definuje dvě základní metody, které jsou v případě vytváření spojeny s HTTP metodou POST a u modifikace s HTTP metodou PUT.

Oba koncové body přejímají jako parametr objekt typu `Task`, který je deserializací získán z těla HTTP požadavku. Kromě něj je možné ale za stejný parametr dosadit i objekt třídy `HpcTask`, který rozšiřuje třídu `Task` o další atributy vztahující se k nastavení prostředí superpočítače. Rozlišení konkrétního typu objektu při deserializaci pak probíhá na základě atributu `targetType` určujícího cílové prostředí a anotace specifikující mapování na konkrétní typ. Podobu anotace s jejími parametry je možné vidět ve výpisu 3.3.

```
@JsonSubTypes({
        @JsonSubTypes.Type(value = Task.class, name = "LOCAL"),
        @JsonSubTypes.Type(value = Task.class, name = "REMOTE"),
        @JsonSubTypes.Type(value = HpcTask.class, name = "HPC")
})
```

Výpis 3.3: Anotace pro specifikaci mapování podle prostředí

Kromě mapování těla požadavku probíhá u metod koncových bodů i jejich validace. O tu se stará validační knihovna zprostředkovaná aplikačním rámcem Spring Boot, která na základě speciálních anotací umístěných v modelu rozhodne o validitě objektu.

V tělech obou metod můžeme dále nalézt volání metod služby `TaskService` realizující samotné operace. V případě vytváření je volána metoda `create`, které je předán objekt reprezentující danou úlohu. Ten je v servisní metodě namapován na příslušný doménový objekt, u něhož je nastaven počáteční stav úlohy a jenž je následně uložen do databáze zavoláním metody `save` na objektu třídy `TaskRepository`.

Obdobným způsobem je řešena i aktualizace, u které je volána metoda `update` s argumentem objektu úlohy. V jejím těle ale již nedochází k přímému mapování úlohy na doménový objekt, avšak místo toho je zde načten původní objekt z databáze, u něhož jsou aktualizovány pouze jeho určité atributy. Tímto způsobem je zabráněno nechtěné nebo nepovolené aktualizaci hodnot.

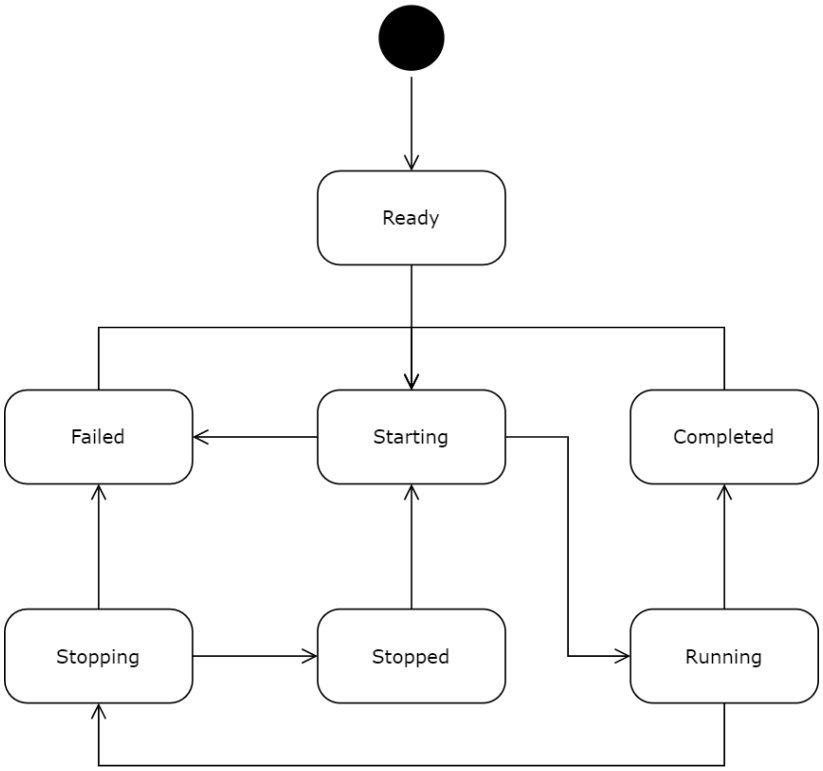
Po vytvoření nebo modifikaci úlohy je dále možné vytvářet nebo upravovat dodatečnou konfiguraci, která zahrnuje správu doplňkových souborů úlohy nebo specifikaci parametrů. K tomu v aplikaci slouží koncové body definované třídami `TaskFileController` a `ParameterController`, jejichž implementace je podobná jako u třídy `TaskController`.

3.6.2 Životní cyklus úloh

Životní cyklus je u každé z úloh řízen jejím stavem, který určuje, v jaké fázi se aktuálně nachází a jaké operace je možné s ní provádět. Každá z úloh proto disponuje stavovým atributem, jehož datový typ je reprezentován výčtovou třídou `TaskState`. Ta definuje jako své položky všechny dostupné stavy spolu s možnými cílovými stavy, které jsou následně využívány pro potřeby validace přechodů mezi nimi.

Na stavy a jejich změny mohou být v aplikaci dále navázány i některé automatizované akce, díky čemuž je možné každou z úloh řídit pouze změnami jejího stavu. Toho je například využito pro spouštění a zastavování úloh, kdy změnou do stavu s názvem STARTING nebo STOPPING je možné provést kroky vedoucí k dokončení patřičné akce.

Přehled včetně vzájemných přechodů je možný vidět v stavovém diagramu znázorněném na obrázku 3.2. Podrobný popis je uveden v následujících částech.



Obrázek 3.2: Stavový diagram

Stav READY

Stav READY představuje počáteční stav úlohy, která dosud nebyla spuštěna. V tomto stavu se automaticky nachází každá nově vytvořená úloha. Jelikož se jedná o prvotní stav, není možné se do něj zpětně dostat z jiného stavu. Naopak z něj se může úloha dostat pouze do stavu STARTING pomocí jejího spuštění.

Stav STARTING

Stav STARTING signalizuje úlohu, u níž bylo zažádáno o její spuštění. Během trvání tohoto stavu dochází teprve k přípravě prostředí což zahrnuje proces kopírování potřebných souborů a spuštění aplikace představující danou úlohu. Požadavek na spuštění úlohy a tím i přechod do tohoto stavu může provést pouze uživatel ze svého rozhraní. Zároveň platí, že do daného stavu se může úloha dostat pouze ze stavů READY, FAILED, STOPPED nebo COMPLETED.

Stav RUNNING

Stavem RUNNING je reprezentována aktivně běžící úloha. Do něj se může dostat po zahájení a spuštění úlohy v cílovém prostředí, kdy klientská aplikace oznámí své spuštění řídícímu serveru právě pomocí změny svého stavu. Přechod do daného stavu je možný pouze ze stavu STARTING.

Stav COMPLETED

Stavem COMPLETED je signalizováno úspěšné dokončení úlohy. O přechod do daného stavu se stará buď tzv. sledovací skript, který kontroluje stav úlohy v cílovém prostředí, nebo v případě lokálního prostředí přímo řídící server. Přechod do něj je možný pouze ze stavu RUNNING.

Stav STOPPING

Stav STOPPING signalizuje úlohu, u níž bylo zažádáno o její zastavení. Během trvání tohoto stavu dochází k ukončování aplikace v cílovém prostředí a k odstranění všech pracovních složek a dočasných souborů. Do tohoto stavu je možné se dostat pouze na žádost uživatele, a to ze stavu RUNNING.

Stav STOPPED

Stavem STOPPED je reprezentována předčasně zastavená úloha, která je znovu dostupná pro další spuštění. Přechod do daného stavu je možný pouze ze stavu STOPPING.

Stav FAILED

Stavem FAILED je u úlohy signalizována chyba, která nastala při požadavku na spuštění nebo zastavení. Do daného stavu je proto možné se dostat pouze ze stavu STARTING nebo STOPPING.

3.6.3 Změna stavu úlohy

Jak je vidět ve výše uvedeném popisu, změny mezi konkrétními stavy jsou obvykle vykonávány rozdílnými entitami. Pro změnu stavu úlohy proto byly v aplikaci vytvořeny dva koncové body rozhraní REST, kde každý z nich podporuje jinou množinu přechodů a vyžaduje jiný typ ověření.

První z takových koncových bodů spolu s jeho mapováním na konkrétní metodu se nachází ve třídě `TaskController` a je primárně určen pro volání z uživatelského rozhraní. Tomu odpovídá i množina podporovaných přechodů, které zahrnují pouze cílové stavy STARTING a STOPPING. Tyto stavy následně reprezentují požadavky na spuštění a zastavení výpočetní úlohy jejichž zpracování je realizováno voláním metody `updateStateAndExecute` třídy `TaskService`. Této metodě jsou kromě parametrů identifikátoru úlohy a cílového stavu předávány i volitelné parametry zahrnující přihlašovací údaje pro připojení ke vzdálenému uzlu. Získání těchto parametrů je realizováno v metodě koncového bodu, a to pomocí mapování na odpovídající hlavičky HTTP požadavku.

V případě druhého koncového bodu je změna stavu možná bez omezení (musí být validní) a jeho odpovídající metoda je implementována třídou `InternalTaskController`, která vyžaduje pro ověření speciální API klíč. Uvnitř této metody se následně kromě ověření API klíče nachází i volání metody `updateState` třídy `TaskService`, které je předán identifikátor a požadovaný stav úlohy získaný mapováním z HTTP požadavku.

3.6.4 Implementace služby TaskRuntimeService

Služba `TaskRuntimeService` má na starost orchestraci všech kroků týkajících se základních operací spuštění a zastavení výpočetních úloh. Pro jejich vykonání vystavuje tato služba dvě metody, jejichž signatury jsou uvedeny ve výpisu 3.4.

```
void startTask(UUID taskId, TaskTargetType targetType, String username, KeyPair
    keyIdentity);
void stopTask(UUID taskId, TaskTargetType targetType, String username, KeyPair
    keyIdentity);
```

Výpis 3.4: Signatury metod rozhraní TaskRuntimeService

U obou metod můžeme vidět stejný výčet parametrů zahrnující identifikátor úlohy, typ cílového prostředí, uživatelské jméno a klíč pro přihlášení ke vzdálenému uzlu. V případě, že cílové prostředí nevyžaduje údaje pro přihlášení, mohou poslední dva parametry nabývat hodnoty *null*.

Obě metody zároveň umožňují asynchronní vykonávání, díky kterému nedochází k blokování volajícího vlákna. Toho je využito při vyřizování požadavků na změnu stavu zahrnujících přechod do stavů STARTING nebo STOPPING, které jsou se spuštěním a zastavením úlohy svázány. Pro docílení asynchronního chování byla využita anotace `@Async`, která je podporována přímo aplikačním rámcem Spring Boot. Deklarací této anotace nad příslušnou metodou je pak zajištěno její zaobalení do proxy objektu, který volání cílové metody provede na jiném vlákně.

Součástí implementace metod je kromě kroků vedoucích ke spuštění nebo zastavení úlohy i využití mechanismů pro synchronizaci přístupu více vláken ke sdílenému prostředku jímž je samotná úloha. Tuto synchronizaci je nutné zajistit především z toho důvodu, aby bylo zabráněno jednak nekonzistentnímu stavu prostředí, ve kterém mají být úlohy spouštěny, ale také kvůli zabránění chyb spojených s vícenásobným spuštěním cílové aplikace.

Synchronizace je v metodách zajištěna pomocí bloku `synchronized` jímž je obalen veškerý kód. Jako zdroj objektů pro synchronizaci byla zvolena mapa `ConcurrentReferenceHashMap` umožňující využití tzv. slabých odkazů (*Weak reference*) pro objekty klíčů a hodnot. Ty reprezentují odkazy, které nejsou sledovány komponentou Grabage Collector, díky čemuž může dojít k odstranění odkazovaných objektů za předpokladu, že nejsou dosažitelné z některé části aplikace pomocí jiného způsobu. Cílem využití této vlastnosti je proto zajištění toho, aby v případě uvolnění všech zámků a tím i synchronizačních objektů došlo k jejich automatickému odstranění z paměti.

3.6.5 Spouštění úloh

Spouštění úloh je v aplikaci realizováno metodou `startTask` třídy `TaskRuntimeService` a skládá se z posloupnosti kroků vedoucích k přípravě cílového prostředí a následnému spuštění aplikace reprezentující danou výpočetní úlohu. Každý z těchto kroků následně tvoří tři fáze, jejichž přehled je uveden níže:

1. Inicializace úlohy
    (a) Změna stavu úlohy na stav STARTING
    (b) Vymazání PID úlohy a monitorovacího skriptu
2. Příprava cílového prostředí
    (a) Získání seznamu souborů spojených s úlohou
    (b) Vytvoření pracovního adresáře
    (c) Vytvoření adresáře pro výstupní soubory úlohy
    (d) Nakopírování souborů spojených s úlohou
3. Spuštění úlohy
    (a) Získání konfigurace úlohy a prostředí
    (b) Spuštění aplikace reprezentující úlohu v cílovém prostředí
    (c) Nastavení PID úlohy a monitorovacího skriptu

S fází inicializace úlohy je jako první krok spojena změna stavu úlohy, a to na stav STARTING. Pomocí něho je signalizován probíhající proces spouštění a je ním i případně zabráněno v další manipulaci s úlohou. Druhým a zároveň posledním krokem této fáze je pak vymazání identifikátoru procesu úlohy a monitorovacího skriptu tak, aby nemohlo dojít k jejich chybnému využití při zastavení úlohy. Ne každá úloha totiž musí využít možnost uložení hodnoty PID pro monitorovací skript, a proto je tato hodnota preventivně mazána.

Jako další fáze následuje příprava cílového prostředí. Ta začíná získáním seznamu souborů spojených s úlohou, které jsou v pozdějším kroku využity pro jejich nahrání do cílového prostředí. Mezitím ale ještě následuje vytvoření pracovního adresáře a adresáře pro výstupní soubory úlohy. První z těchto adresářů slouží pro uchování všech souborů a složek týkajících se dané úlohy a jako její umístění je u prostředí využívajících vzdálené uzly zvolen domovský adresář uživatele. Pro její pojmenování je pak použita řetězcová podoba identifikátoru úlohy, kterým je zajištěna unikátnost dané složky na cílovém zařízení. Druhý z adresářů je následně vytvořen v tom pracovním a slouží pro uchování výstupních souborů vygenerovaných úlohou, které mají být po jejím dokončení automaticky nahrány zpět na server. Ve všech výše uvedených případech zajišťuje práci se souborovým systémem cílového prostředí komponenta implementující jednotné rozhraní `TaskRunner`.

Poslední fáze je tvořena třemi kroky a začíná získáním konfigurace úlohy a prostředí z databáze. Reprezentace této konfigurace je zajištěna třídou `TaskConfig`, která v sobě obsahuje například atributy pro uložení parametrů úlohy a prostředí nebo název spustitelného souboru. Objekt této třídy je následně využit ve druhém kroku při spuštění aplikace reprezentující úlohu v cílovém prostředí. K tomu je stejně jako ve fázi přípravy využita odpovídající komponenta implementující

rozhraní `TaskRunner`, která poskytuje jednotný přístup pro práci v různých typech prostředí. Spuštění je v tomto případě provedeno metodou `runTask`, která po svém dokončení vrací objekt typu `JobDescriptor`. Ten je složen z dvojice atributů určených k uložení hodnot identifikátorů procesu úlohy a monitorovacího skriptu, které slouží k případnému ukončení celé úlohy. Objekt tohoto typu je následně využit v dalším a zároveň posledním kroku, kde jsou jeho hodnoty uloženy do databáze pro pozdější využití.

Po dokončení poslední fáze je výpočetní úloha považována za spuštěnou, avšak o finálním přechodu úlohy do stavu RUNNING, tedy běžící, rozhoduje až sama aplikace. K signalizaci takového stavu je využita klientská knihovna, která v rámci své implementace umožňuje zaslat potřebný požadavek řídícímu serveru.

3.6.6 Zastavování úlohy

Zastavení úlohy je realizováno metodou `stopTask` třídy `TaskRuntimeService` a skládá se z několika kroků vedoucích k zastavení aplikace reprezentující danou úlohu a následnému vyčištění prostředí. Přehled těchto kroků je uveden níže:

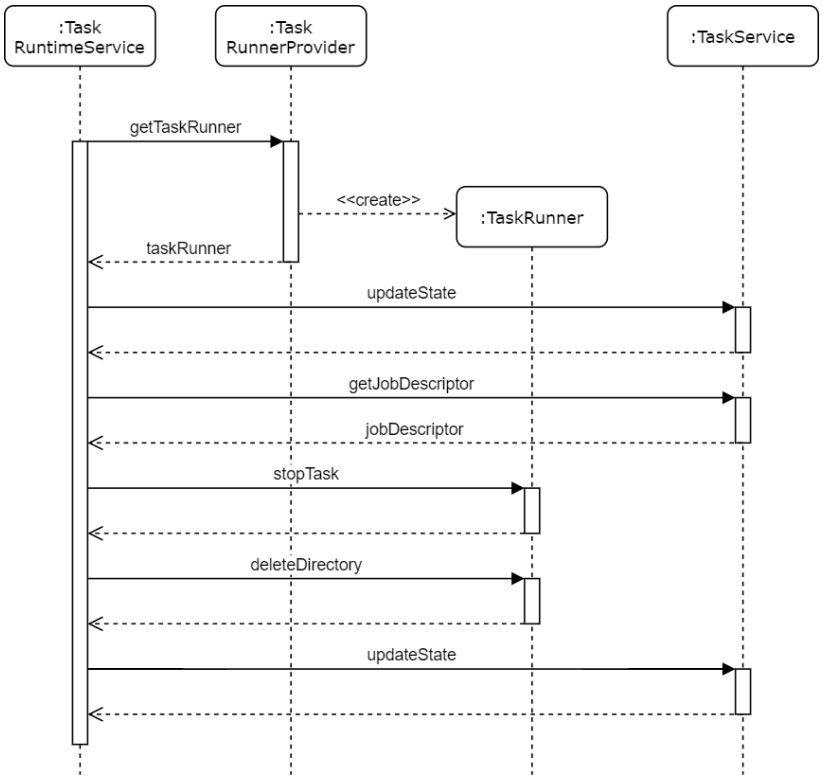
1. Změna stavu úlohy na stav STOPPING
2. Získání PID úlohy a monitorovacího skriptu
3. Zastavení procesu monitorovacího skriptu (je-li využit)
4. Zastavení aplikace reprezentující výpočetní úlohu
5. Odstranění pracovní složky projektu
6. Změna stavu úlohy na stav STOPPED

Jedním z prvních kroků celého procesu je změna stavu úlohy na stav STOPPING. Tím je obdobně jako v případě spouštění úlohy signalizováno její probíhající zastavování, během něhož s ní nesmí být možné dále manipulovat. V dalším kroku poté přichází na řadu získání identifikátorů procesu úlohy a monitorovacího skriptu z databáze, které jsou zaobaleny do jediného objektu typu `JobDescriptor`. Ten je dále předán metodě `stopTask` komponenty `TaskRunner`, která má za úkol zajistit postupné zastavení procesů monitorovacího skriptu (je-li využit) a samotné aplikace reprezentující danou výpočetní úlohu.

Po úspěšném ukončení obou procesů je zahájeno odstranění pracovní složky projektu v cílovém prostředí. K tomu je využita metoda `deleteDirectory` příslušné komponenty `TaskRunner`, které je předán identifikátor úlohy představující název složky určené ke smazání.

V posledním kroku je následně realizován přechod úlohy do stavu STOPPED, který signalizuje ukončenou úlohu. Na základě tohoto stavu může být poté úloha opětovně spuštěna.

Celý proces zastavení úlohy v prostředí superpočítače je zachycen sekvenčním diagramem jehož podobu lze vidět na obrázku 3.3.



Obrázek 3.3: Sekvenční diagram zastavení úlohy

3.7 Komponenty pro spouštění a řízení úloh v cílovém prostředí

Spouštění a řízení úloh v cílových prostředích je v aplikaci zajištěno pomocí tří komponent, které odpovídají prostředí lokálnímu, vzdálenému a prostředí superpočítače. Každá z těchto komponent implementuje jednotné rozhraní `TaskRunner` nabízející několik obecných metod určených pro manipulaci s úlohou a jejími soubory. Výčet těchto metod je uveden ve výpisu 3.5.

```
JobDescriptor runTask(TaskConfig taskConfig);
void stopTask(JobDescriptor jobDescriptor);
boolean isTaskAlive(String processId);
void copyFiles(Set<FileCopyDescriptor> files);
void createDirectory(String dir);
void deleteDirectory(String dir);
TaskTargetType appliesTo();
```

Výpis 3.5: Signatury metod rozhraní TaskRunner

S využitím rozhraní `TaskRunner` pro přístupu k jednotlivým komponentám se můžeme setkat zejména u služby `TaskRuntimeService`, která díky němu nemusí u každé operace rozlišovat prostředí, ve kterém má být provedena.

3.7.1 Konfigurace komponent

Konfigurace komponent pro řízení úloh je zajištěna pomocí několika modelů, které jsou rozděleny podle oblasti určení a cílového prostředí. V základním přehledu všech tříd představujících tyto modely můžeme najít například třídy:

- SshConfig
- TaskParamsConfig
- RemoteRunnerConfig
- LocalRunnerConfig

S konfigurační třídou `SshConfig` se můžeme setkat zejména u komponent využívajících pro přístup ke vzdálenému prostředí SSH připojení. Zde jsou využity její atributy pro definici adresy serveru, uživatelského jména a přístupového klíče. Inicializace hodnot je obvykle prováděna ze dvou zdrojů. Prvním z nich je konfigurační soubor aplikace, kde jsou definována společná nastavení týkající se například adresy serveru pro jednotlivá prostředí. Jako druhý a zároveň doplňující zdroj jsou využity údaje předané uživatelem, které se vážou k uživatelskému jménu a přístupovému klíči.

Druhá z konfiguračních tříd `TaskParamsConfig` je dále určena pro definici tzv. systémových parametrů úloh. Ty aktuálně zahrnují pouze jedinou položku, která je představována atributem `managementServer`. Hodnotou tohoto atributu je vyjádřena adresa řídícího serveru, která je v úlohách využívána pro komunikaci s ním.

Poslední dvě třídy jsou rozdělené podle typu prostředí a jsou zaměřené na agregaci konfigurace reprezentované některými z předchozích tříd. Kromě těchto atributů přidává tato třída i vlastní atribut `workingDir`, který je určen pro definici pracovní složky. Ta je využita jako základní cesta například v komponentách pro spouštění a řízení úloh. Inicializace tohoto atributu je prováděna prostřednictvím uživatelského jména sloužícího pro přihlášení k serveru skrze SSH nebo v případě lokálního prostředí pomocí hodnoty načtené z konfiguračního souboru aplikace.

3.7.2 Poskytovatel komponent typu TaskRunner

S přihlédnutím k možnostem konfigurace jednotlivých komponent určených pro řízení úloh v cílovém prostředí byla do aplikace přidána služba `TaskRunnerProvider` umožňující vytváření jejich instancí. Hlavním cílem tohoto poskytovatele je z předaných parametrů a za využití globální konfigurace vytvořit instance tříd implementující rozhraní `TaskRunner`, které by byly spravovány aplikačním rámcem Spring Boot a umožňovaly by jejich pozdější využití ve třídě `TaskRuntimeServiceImpl`.

K tomu je využita parametrizovaná varianta návrhového vzoru Tovární metody (*Factory method*), jejíž cílem je poskytnout rozhraní pro tvorbu objektů určitého druhu, který je zvolen na základě předaného parametru. Současně také platí, že tento druh objektů musí implementovat stejné rozhraní. [20]

Ve třídě `DefaultTaskRunnerProvider` je tato tovární metoda představována metodou `getTaskRunner`, která jako určující parametr pro vytváření objektů využívá typ cílového prostředí. O samotné vytváření instancí se v této třídě stará dvojice objektů typu `ObjectProvider<T>` s typovými parametry odpovídající třídám `SshRemoteTaskRunner` a `HpcTaskRunner`.

Součástí rozhraní `ObjectProvider<T>` je několik metod sloužících pro získání instance na některou z registrovaných *Bean* komponent odpovídající typovému parametru `T`. Každá z těchto *Bean* komponent může disponovat různým rozsahem platnosti definovaným anotací `@Scope`, která mimo jiné určuje i kdy má dojít k jejímu vytvoření a zániku. V případě tříd `SshRemoteTaskRunner` a `HpcTaskRunner` byl pro rozsah platnosti využit typ *Prototype*, díky kterému vrací metoda `getObject` komponenty `ObjectProvider` pokaždé jejich novou instanci.

Samotné vytváření těchto instancí je realizováno pomocí konstruktoru, jehož parametry odpovídají argumentům předaných během volání metody `getObject`. U obou výše uvedených tříd byl zvolen konstruktor přejímající objekty typu `SshConfig` a `RemoteRunnerConfig`, které jsou vytvářeny na základě argumentů předaných metodě `getTaskRunner`. Ty v tomto případě zahrnují atribut `username` a `keyIdentity`, které slouží pro přihlášení ke vzdálenému serveru a k specifikaci pracovního adresáře. Kromě přímé inicializace atributů skrze konstruktor je k dispozici i inicializace za využití mechanismu *Dependency Injection*, kterým mohou být získány odpovídající instance objektů z aplikačního kontextu. Pro jeho využití stačilo pouze u zvolených atributů uvést anotaci `@Autowired`.

3.7.3 Komponenta HpcTaskRunner

Spouštění a řízení úloh v prostředí superpočítače je v aplikaci zajištěno pomocí komponenty `HpcTaskRunner`, která pro tyto účely implementuje rozhraní `TaskRunner`. V rámci ní jsou nejdůležitější její první dvě metody `runTask` a `stopTask` sloužící pro spuštění a zastavení úlohy. Součástí implementace obou z metod je řada kroků vedoucích k jejich úspěšné realizaci v prostředí superpočítače. V případě metody `runTask` jsou tyto kroky rozděleny do následujících fází:

1. Příprava úlohy
   (a) Sestavení příkazu vytvářejícího požadavek na spuštění úlohy
   (b) Sestavení příkazu pro spuštění aplikace reprezentující danou úlohu
   (c) Sestavení příkazu pro spuštění monitorovacího skriptu
2. Tvorba skriptů
   (a) Vytvoření skriptu pro dávkové spuštění úlohy na přidělených uzlech (job script)
   (b) Vytvoření skriptu pro inicializaci a spuštění konkrétní instance úlohy na daném uzlu (task script)

(c) Vytvoření monitorovacího skriptu

3. Vytvoření požadavku na spuštění úlohy

4. Spuštění monitorovacího skriptu

S počáteční fází je spojena příprava úlohy, která je složena z kroků sestavující postupně tři příkazy. První z nich je určen pro vytvoření požadavku na spuštění úlohy a pro jeho sestavení jsou využity údaje předané pomocí parametru `taskConfig`. Tyto údaje jsou v příkazu využity jako parametry utility `qsub`, která na základě nich provede žádost o alokaci potřebných zdrojů a úlohu zařadí do fronty pro budoucí spuštění.

Druhý z příkazů má na starost spuštění aplikace reprezentující úlohu a je složen z volání aplikace `java`, které jsou předány tzv. systémové a uživatelsky definované parametry spolu s cestou k spustitelnému souboru. Poslední z příkazů dále slouží ke spuštění monitorovacího skriptu. U toho je vyžadován běh na pozadí tak, aby mohlo být po dokončení všech kroků nutných ke spuštění úlohy spojení se serverem ukončeno. To je zajištěno direktivou reprezentovanou symbolem *"&"* spolu se zavoláním příkazu `nohup`, který zabraňuje ukončení skriptu v případě, že je ukončen příslušný terminál, jímž byl spuštěn. Pro pozdější ukončení skriptu je proto zvolen způsob využívající identifikátor procesu, který je získán pomocí utility `echo`, jíž je předána druhá direktiva označená jako *"$!"*.

Další z fází je sestavena z tvorby skriptů, které jsou pro běh a řízení úlohy nezbytné. K tomu jsou využity již předpřipravené šablony skriptů umístěné ve složce *src/main/resources/scripts/hpc*, které obsahují zástupné řetězce určené pro jejich nahrazení odpovídajícími hodnotami. V aplikaci jsou tyto šablony načítány při jejím spuštění, a to pomocí třídy `ResourceLoader` poskytované aplikačním rámcem Spring Boot. Díky ní je pro každou šablonu získán odpovídající vstupní proud binárních dat, který je pomocí třídy `InputStreamReader` převeden na proud znaků s kódováním UTF-8. Z proudu znaků jsou poté za využití obalující třídy `BufferedReader` přečteny jednotlivé řádky, které jsou následně spojeny oddělovačem řádků *"\n"*. Provedení konverze znaků na kódování UTF-8 a operace čtení po řádcích s jejich následným spojením, bylo v aplikaci nutné provést zejména z důvodu zajištění přenositelnosti obsahu skriptů mezi prostředím serveru, na kterém poběží řídící server a prostředím superpočítače. [21]

Načtené šablony skriptů jsou následně využívány komponentou `HpcScriptTemplateResolver`, která umožňuje jejich další zpracování pomocí tří vystavených metod. Ty jsou postupně volány v metodě `runTask`, kde je jejich výsledek v podobě souboru zapsán do pracovního adresáře úlohy na serveru.

Mezi skripty, které jsou tímto způsobem zpracovány patří například skript realizující dávkové spuštění úlohy na přidělených uzlech. V něm se můžeme následně setkat s využitím druhého ze skriptů určeného ke spuštění aplikace reprezentující úlohu na přiděleném uzlu pro nějž byla stejně jako u prvního skriptu využita implementace popsaná v diplomové práci *Modul RBM a DBM pro program Modeler neuronových sítí* [22]. Jeho součástí je kromě příkazu sestaveného v první fázi

i volání utility `ssh`, která slouží pro přesměrování komunikace mířící přes místní port uzlu 6000 na port 6000 přihlašovacího uzlu. Takové přesměrování je pak možné využít s druhým přesměrováním vytvořeným mezi superpočítačem a řídícím serverem, kterému je možné prostřednictvím tohoto komunikačního kanálu zasílat metriky nebo jiné dodatečné informace. To může být užitečné zejména v době ladění nebo v případě, že řídící server není dostupný z venkovní sítě. Úplná podoba skriptu je vidět ve výpisu 3.6.

```
#!/bin/bash
module load Java/11.0.2
$SET_WORKING_DIRECTORY_COMMAND
ssh -f -o ExitOnForwardFailure=yes -L 6000:localhost:6000 login2 sleep 15
$TASK_EXECUTION_COMMAND
```

Výpis 3.6: Šablona skriptu pro spuštění aplikace reprezentující úlohu

Na jednotlivých řádcích si můžeme všimnout i využití zástupných řetězců začínajících symbolem *"$"* nebo volání příkazu sloužícího pro načtení modulu Javy.

Poslední ze skriptů slouží pro sledování běhu úlohy za účelem realizace dokončovacích akcí. Sledování úlohy je v tomto případě realizováno pomocí periodické kontroly pracovního adresáře, ve kterém je každých 10 sekund ověřována přítomnost souboru obsahujícího standardní výstup úlohy. Ten je po dokončení úlohy generován plánovačem úloh a v případě superpočítače Barbora má formát *".oXXXXXX"*, kde symbol *"X"* představuje číslici. Důvodem volby tohoto přístupu založeného na využití souborového systému namísto specializované utility `qstat`, bylo především snížení zátěže plánovače úloh.

V pořadí třetí a čtvrtá fáze je tvořena pouze spuštěním příkazů vytvořených ve fázi první. Po vykonání každého z nich jsou ze standardního výstupu získány celkem dva řetězce představující identifikátor úlohy spuštěné superpočítačem a identifikátor procesu monitorovacího skriptu. Ty jsou následně použity pro vytvoření objektu typu `JobDescriptor`, který je metodou `runTask` nakonec vrácen.

Kapitola 4

Klientská knihovna

Tato kapitola se věnuje implementaci klientské knihovny. Kromě základních tříd je zde popsána implementace funkcí spojených se záznamem metrik, nahráváním a stahováním souborů, naslouchání změnám v parametrech nebo zpracováním požadavků na vyhodnocení vstupních dat.

4.1 Implementace rozhraní

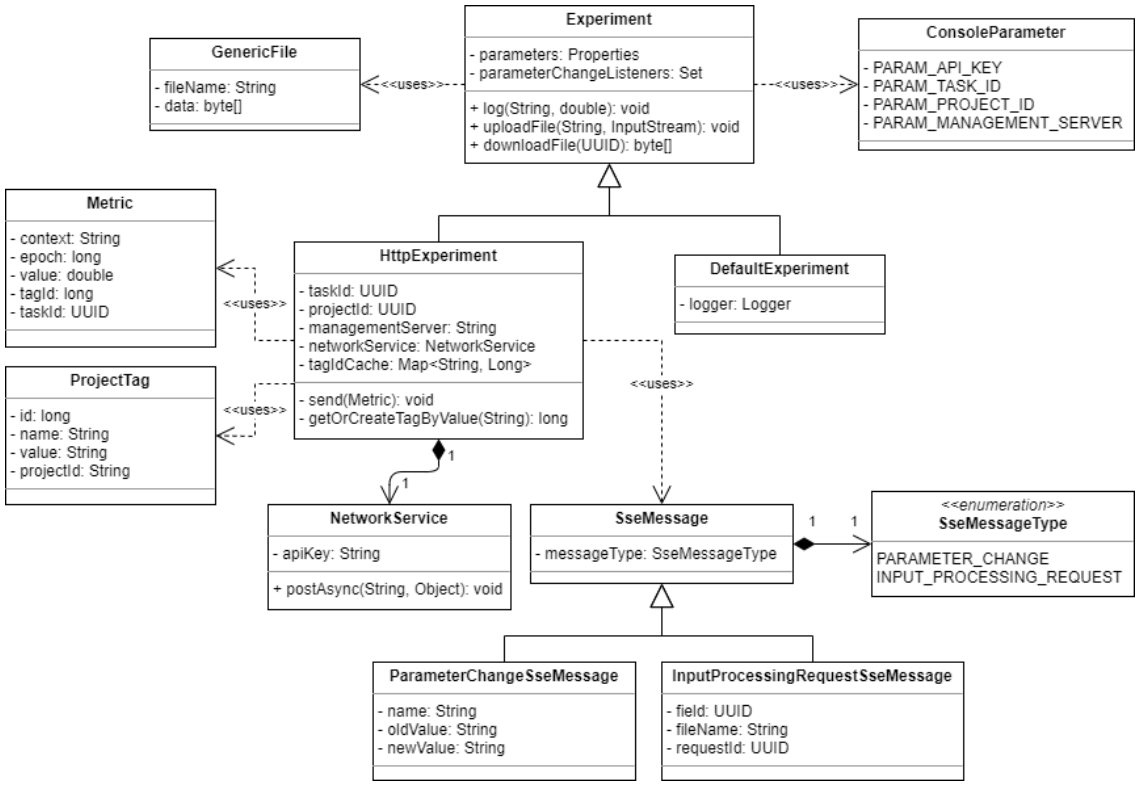
Hlavním cílem klientské knihovny je umožnit koncovým aplikacím záznam metrik na řídící server, reportování průběhu výpočtu nebo získání konfiguračních parametrů. K tomuto účelu vystavuje knihovna prostřednictvím abstraktní třídy `Experiment` rozhraní, které poskytuje jednotný přístup ke všem jejím funkcím.

Ty mohou být realizovány prostřednictvím tříd `HttpRemoteExperiment` a `DefaultExperiment`, které abstraktní třídu `Experiment` rozšiřují a umožňují knihovnu využívat jak ve spojení s řídícím serverem, tak i samostatně mimo něj. Vytvoření potřebné instance a získání přístupu k ní je v knihovně realizováno prostřednictvím návrhového vzoru Jedináček (*Singleton*). Cílem tohoto vzoru je zajistit, aby pro danou třídu mohla existovat pouze jediná její instance s možností globálního přístupu k ní. [20]

Pro tyto účely je v abstraktní třídě `Expirement` vystavena statická metoda `getInstance`, která umožňuje získat odkaz na instanci jedné z jejích podtříd. Ten je ve třídě `Experiment` uchováván v podobě privátního statického atributu a o volbě kterým konkrétním podtypem bude inicializován je rozhodováno na základě prostředí, v němž je aplikace spuštěna. Takovým prostředím může být lokální prostředí určené pro vývoj nebo prostředí spravované řídícím serverem. Jejich rozlišení je prováděno na základě přítomnosti povinných parametrů vyžadovaných pro správu úlohy řídícím serverem.

V případě lokálního prostředí určeného pro vývoj aplikace vrací metoda `getInstance` instanci třídy `DefaultExperiment`, která ve své implementaci umožňuje u všech metod využít pouze záznam základních informací do konzole. Jedná se tedy pouze o pomocnou a dočasnou třídu pro ladění apli-

kace. Naopak v případě spuštění aplikace prostřednictvím řídícího serveru je metodou `getInstance` vrácena instance třídy `HttpRemoteExperiment`, která již plně realizuje veškerou funkcionalitu týkající se záznamu metrik a dalších rozšiřujících funkcí. S ní je spojena i řada dalších tříd, jejichž přehled je uveden v třídním diagramu, který je vidět na obrázku 4.1.



Obrázek 4.1: Přehled základních tříd klientské knihovny

4.2 Implementace základních funkcí pro záznam metrik

Základní funkce pro záznam metrik na řídící server jsou v knihovně poskytovány prostřednictvím čtyř metod s názvem `log`, které jsou implementovány ve třídě `HttpRemoteExperiment`. Každá z těchto metod se liší ve svém počtu parametrů a jejich typech, které dohromady poskytují několik možných variant záznamu metrik. Přehled signatur přetížených metod je vidět ve výpisu 4.1.

```
public void log(String context, long epoch, String tag, double value);
public void log(String context, String tag, double value);
public void log(long epoch, String tag, double value);
public void log(String tag, double value);
```

Výpis 4.1: Signatury metod určené pro záznam metrik

U těchto signatur je možné si všimnout dvou povinných parametrů `tag` a `value` představujících název metriky a její hodnotu. Kromě nich je možné si dodatečně při záznamu metrik zvolit i řetězcové pojmenování kontextu nebo spolu s metrikou zaznamenat i číslo epochy, ve kterém byla změřena. Oba z těchto parametrů je pak možné využít při sestavování grafických vizualizací, kde mohou poskytnout upřesňující údaje o metrikách.

Implementace výše uvedených metod je dále složena z několika částí, přičemž první z nich je tvořena získáním identifikátoru tagu. Ten je vyžadován řídícím serverem, který jej namísto jeho řetězcové podoby využívá pro správné spárování metrik a jejich naměřených hodnot. Získání identifikátoru pro každý tag probíhá za využití mezipaměti nebo koncového bodu řídícího serveru. V případě že identifikátor tagu není nalezen v mezipaměti následuje jeho hledání v databázi pomocí koncového bodu řídícího serveru. Pokud není nalezen ani tam, je provedeno vytvoření nového záznamu tagu spolu s následným vrácením jeho identifikátoru. Ten je pak uložen do mezipaměti, ze které je při příští příležitosti načten.

V poslední části metod následuje vytvoření obalujícího objektu typu `Metric`, který je určen pro přenos všech uvedených parametrů na řídící server. Jeho odeslání je prováděno asynchronně pomocí metody `postAsync` volané na objektu třídy `NetworkService`. Díky tomu nedochází k blokování volajícího vlákna a tím ke zbytečnému zpomalování výpočtu.

4.3 Implementace rozšiřujících funkcí

Kromě funkcí pro základní záznam metrik je knihovna vybavena i několika rozšiřujícími funkcemi. Ty zahrnují například možnost nahrávání či stahování souborů, naslouchání změnám v parametrech nebo možnost reagovat na požadavky pro zpracování vstupních dat. Podrobnější popis jejich implementace je uveden v následujících částech.

4.3.1 Nahrávání a stahování souborů

Pro nahrávání souborů na řídící server nabízí knihovna dvě přetížené metody s názvem `uploadFile`. Jejich signatury obsahují dva povinné parametry `name` a `inputStream`, které představují název a vstupní proud dat. Kromě nich je v jedné z přetížených variant přítomen i volitelný parametr pro specifikaci popisu souboru. Ve výpisu 4.2 je vidět jejich přehled.

```
public void uploadFile(String name, String description, InputStream inputStream);
public void uploadFile(String name, InputStream inputStream);
```

Výpis 4.2: Signatury metod určené pro nahrávání souborů

Nahrávání dat následně probíhá za využití třídy `NetworkService`, která pro tyto potřeby umožňuje sestavit a odeslat HTTP požadavek typu *"multipart/form-data"*. V něm je kromě vstupního

proudu a předaných parametrů specifikován i identifikátor projektu, pomocí kterého je zajištěno správné přiřazení souboru a jeho metadat.

Naopak pro stažení souboru nahraného k projektu je určena metoda `downloadFile`, která přebírá jediný parametr v podobě jeho identifikátoru. Ten je předán stejnojmenné metodě třídy `NetworkService`, která realizuje jeho stažení. Výsledný soubor je pak předán zpět volající metodě v podobě pole typu `byte`.

4.3.2 Naslouchání změnám v parametrech

Parametry úlohy představují jeden z hlavních prostředků pro její konfiguraci. Možnost dynamicky reagovat na jejich změny proto představuje velmi silný nástroj pomocí něhož lze za chodu velice snadno upravovat chování celé aplikace.

Knihovna pro tyto účely nabízí jednoduché rozhraní postavené na modelu Posluchače událostí (*Event-Listener*) [23]. Díky němu je možné k odběru notifikací o změnách v parametrech přihlásit kteroukoliv komponentu, která má o jejich příjem zájem.

Základem tohoto rozhraní je třída `ParameterChangedEvent` zapouzdřující informace o změně. Prostřednictvím atributů této třídy jsou příjemci informování o názvu parametru, u něhož došlo ke změně a o jeho předchozí a nové hodnotě.

K odběru událostí se mohou ostatní komponenty přihlásit pomocí rozhraní vystaveného třídou `Experiment`. V ní jsou pro práci s posluchači událostí implementovány tři metody jejichž signatury jsou vidět ve výpisu 4.3.

```
public void addParameterChangeListener(ParameterChangeListener listener);
public void removeParameterChangeListener(ParameterChangeListener listener);
public ParameterChangeListener[] getParameterChangeListeners();
```

Výpis 4.3: Signatury metod určené pro práci s poluchačem událostí týkající se změn parametrů úlohy

Každý z posluchačů událostí musí implementovat rozhraní `ParameterChangeListener`, které obsahuje deklaraci jediné metody `parameterChanged`. Ta je volána třídou `Experiment` pokaždé když dojde ke změně některého z parametrů úlohy, přičemž jako argument je jí předán objekt typu `ParameterChangedEvent`.

O změnách v parametrech je samotná knihovna informována skrze SSE (*Server-sent events*) události, které jsou asynchronně zasílány řídícím serverem pomocí HTTP protokolu. Stejně jako v případě modelu Posluchače událostí je i zde posluchač ve formě koncové aplikace registrován pro příjem notifikací generovaných řídícím serverem. K tomu je využit patřičný koncový bod, který odběr SSE zpráv umožňuje.

Notifikace jsou pak zasílány úlohám (posluchačům) vždy když dojde ke změně některého z jejich parametrů. Platí tedy, že k odběru notifikací se může pro každou definovanou úlohu přihlásit libovolný počet koncových aplikací s odpovídajícím API klíčem.

Správu spojení a příjem událostí má v knihovně následně na starost třída `NetworkService`.

4.3.3 Zpracování požadavků na vyhodnocení vstupních dat

Možnost vyhodnocení vstupních dat za běhu aplikace je další z funkcí, která může znatelným způsobem usnadnit ladění vyvíjených algoritmů. Pomocí tohoto nástroje tak lze například získat náhled na aktuální stav učení neuronové sítě a tím i přibližně zjistit, zda síť splňuje požadavky na námi zamýšlené využití.

Obdobně jako v předchozím případě zahrnujícím parametry je i pro zpracování požadavků na vyhodnocení vstupních dat využit mechanizmus SSE pro příjem notifikací z řídícího serveru. Pomocí něj jsou klientským aplikacím zasílány žádosti obsahující identifikátor souboru, jeho název a identifikátor požadavku.

Po přijetí každého požadavku je následně řízení zpracování předáno metodě `handleInputProcessingRequest` třídy `HttpRemoteExperiment`. V ní je provedeno stažení vstupních dat v podobě souboru, které jsou poté předány zaregistrovanému posluchači událostí. Ten má za úkol data zpracovat a poskytnout prostřednictvím vráceného objektu typu `GenericFile` výsledek s daty. Výstupní data jsou poté automaticky nahrána prostřednictvím služby `NetworkService` jako soubor do odpovídajícího projektu.

Registrace objektu určeného pro vyhodnocování vstupních dat lze provést pomocí metody `setInputProcessingRequestHandler` nacházející se ve třídě `Experiment`. Každý takový objekt musí implementovat rozhraní `InputProcessingRequestHandler`, které obsahuje jedinou metodu určenou pro specifikaci logiky týkající se zpracování vstupu. Tato metoda je pak volána knihovnou ve chvíli, kdy je přijat nový požadavek a jako argument je jí předán objekt typu `InputProcessingRequest` nesoucí vstupní data.

4.4 Komunikace s řídícím serverem

Komunikace s řídícím serverem je v knihovně zprostředkována skrze třídu `NetworkService`, která ostatním komponentám knihovny nabízí některé základní metody pro vytváření a odesílání HTTP požadavků. Jejich výčet je uveden ve výpisu 4.4.

U několika metod je možné si všimnout přítomnost slova *"Async"*, které označuje ty metody, u nichž je pro odesílání požadavků využit asynchronní přístup. Díky němu je pak zajištěno to, že nedochází k blokování volajícího vlákna.

V implementaci je pro tvorbu a odeslání požadavků převážně využita třída `HttpClient`, která je běžnou součástí jazyka Java od verze 11. Ta je ve spolupráci s dalšími třídami z balíčku

```
public void postAsync(String url, Object object);
public void post(String url, Map<String, String> formData, InputStream inputStream
    );
public void putAsync(String url, Object object);
public <T> Optional<T> putForObject(String url, Object object, Class<T>
    responseType);
public void delete(String url);
public byte[] downloadFile(String url);
public void connectToSseEventSource(String url, Consumer<InboundSseEvent> consumer
    , Consumer<Throwable> errorConsumer);
```

Výpis 4.4: Signatury metod třídy NetworkService

`java.net.http` použita pro sestavení jednoduchých požadavků, které jsou součástí metod jako `putAsync` nebo `delete`. Pro speciální případy vyžadující například práci s tzv. *Multipart* požadavky pro nahrávání souborů nebo práci s SSE je využita knihovna Jersey.

Samotná inicializace objektu třídy `NetworkService` probíhá za využití konstruktoru přebírajícího jediný parametr v podobě API klíče. Ten je využíván pro autentifikaci koncové aplikace a jeho řetězcová hodnota je vkládána do každého požadavku ve formě HTTP hlavičky s názvem *Authorization*.

Kapitola 5

Uživatelské rozhraní

Tato kapitola se zabývá implementací vybraných částí uživatelského rozhraní a popisem technologií, které byly v průběhu jeho vývoje využity.

5.1 Použité technologie

Pro vývoj aplikace grafického uživatelského rozhraní byl jako hlavní jazyk zvolen JavaScript, respektive jeho rozšíření v podobě jazyka TypeScript, která do něj přináší podporu typů. S jeho využitím se můžeme setkat zejména u webových aplikací běžících v prohlížeči, ale v poslední době nachází své uplatnění i při vývoji mobilních aplikací nebo u aplikací běžících na serveru. Mezi důvody, které stály za zvolením tohoto jazyka a obecně formy tzv. jednostránkové aplikace (*Single Page Application*) patří především jeho široké možnosti využití spolu s velkou podporou nejrůznějších knihoven. Díky tomu bylo možné vytvořit komplexní uživatelské rozhraní s nejrůznějšími interaktivními prvky, které usnadňují a zpříjemňují používání aplikace. Přehled použitých knihoven spolu s jejich stručným popisem je uveden níže.

React – flexibilní knihovna určená pro tvorbu grafických uživatelských rozhraní. Jejím základem je přístup založený na využití komponent, které představují nezávislé znovupoužitelné části. Tyto části následně usnadňují tvorbu komplexních uživatelských rozhraní. [24]

React Redux – knihovna určená pro centralizovanou správu stavu v aplikaci. Díky ní je možné modifikovat stav jednotlivých komponent konzistentním a předvídatelným způsobem za pomoci událostí. [25]

Redux-Saga – knihovna umožňující realizaci dodatečných operací spojených se zpracováním událostí. V aplikaci nachází využití zejména při tvorbě asynchronních požadavků na koncové body rozhraní REST, které vystavuje řídící server. [26]

Reactstrap – knihovna usnadňující práci s grafickými komponentami aplikačního rámce Bootstrap v aplikacích využívajících knihovnu React. Spolu s aplikačním rámcem Bootstrap jsou poskytovány již předem připravené nástroje a komponenty zahrnující například ovládací a formulářové prvky. [27]

React router – knihovna poskytující sadu navigačních komponent, které umožňují navigaci mezi stránkami (reprezentovanými komponentami) na základě URL adresy. [28]

5.2 Správa projektů a úloh

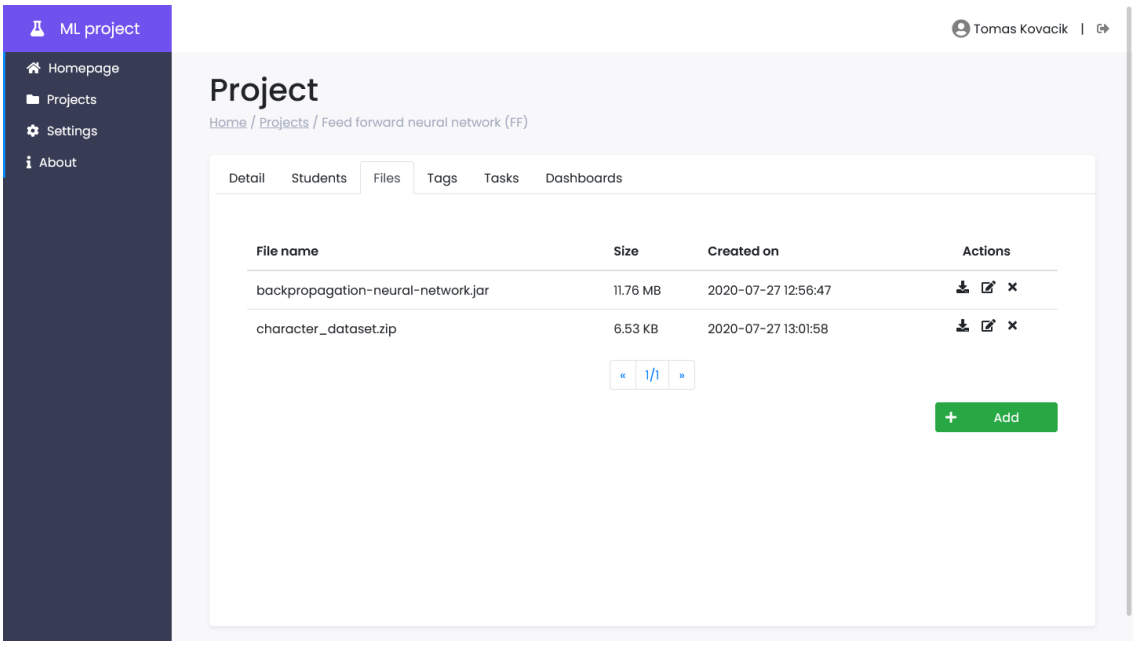
Správa projektů je v aplikaci zajištěna pomocí dvojice kontejnerů `ProjectListContainer` a `ProjectFormContainer`, které umožňují uživatelům jejich procházení, vytváření a editaci. Pro účely procházení projektů definuje první z kontejnerů tabulku tvořenou komponentou `TableView`, do níž jsou předány objekty reprezentující záznamy projektů. Tyto záznamy jsou získávány z řídícího serveru při každé návštěvě stránky s daným kontejnerem prostřednictvím speciálních komponent umístěných ve složce *api* a to na základě role aktuálně přihlášeného uživatele. V případě běžného uživatele jsou tak zobrazeny pouze ty projekty jejichž je uživatel členem. Naopak administrátor může vidět projekty bez omezení.

Spolu s tabulkou je v kontejneru umístěno i tlačítko pro vytvoření nového projektu, které uživatele přesměruje na novou stránku s druhým kontejnerem `ProjectFormContainer`. Ten umožňuje na základě dat zadaných prostřednictvím formuláře vytvořit projekt nový nebo upravit stávající. K tomu jsou opět využity služby komponenty ze složky *api* pro odesílání požadavků na řídící server.

Kromě správy samotných projektů je v aplikaci k dispozici i několik kontejnerů, které umožňují provádět správu jejich dílčích částí týkajících se například přiřazených studentů, projektových souborů, tagů, samotných úloh nebo skupin vizualizací. Každá z těchto částí je následně dostupná pomocí hlavního kontejneru s názvem `ProjectManagementContainer`, který je reprezentován samostatnou stránkou v aplikaci. V něm jsou jednotlivé sekce rozděleny do záložek, které vykreslují obsah odpovídajících kontejnerů. Podrobnému popisu některých z těchto sekcí se dále věnují následující podkapitoly.

5.3 Správa projektových souborů

Pro správu projektových souborů byl v aplikaci implementován kontejner `ProjectFilesTabContainer`, jehož centrem je opět tabulka obsahující seznam všech souborů. Kromě jejich procházení poskytuje kontejner možnost i nahrání souborů nových, a to prostřednictvím tlačítka umístěného v jeho spodní části nebo pomocí přetáhnutí souboru ze složky počítače do oblasti tabulky. V obou případech byla využita knihovna react-dropzone, která po vybrání souboru předává objekt získaného souboru komponentě `ProjectFileFormModal`. Jejím



Obrázek 5.1: Správa projektových souborů

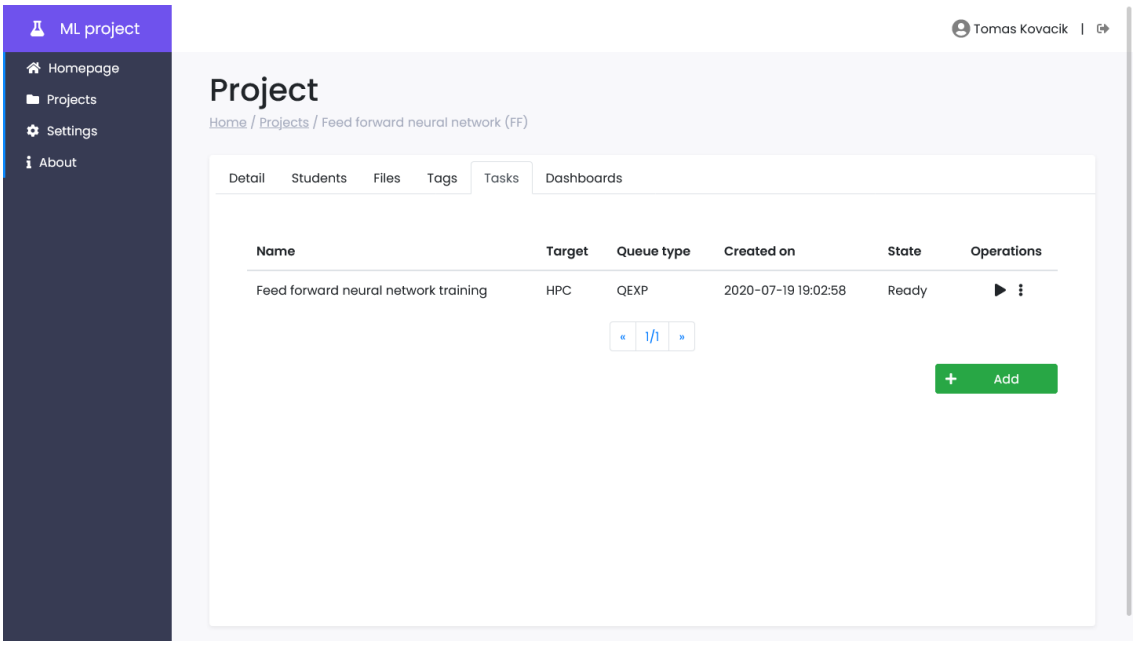
úkolem je pak umožnit uživateli prostřednictvím dialogového okna vyplnit dodatečné informace o souboru zahrnující jeho název a krátký popis.

Nahrání souboru na řídící server je dále řešeno prostřednictvím modulu `FileApi`, který umožňuje vytvořit tzv. *Multipart* požadavek s doplňujícími informacemi o souboru včetně jeho obsahu. Současně jsou v tomto modulu řešeny i další funkce týkající se například stažení, aktualizace nebo smazání souborů. S nimi jsou následně provázány zbývající ovládací prvky a komponenty v uživatelském rozhraní.

Výsledná podoba uživatelského rozhraní pro správu projektových souborů je vidět na obrázku 5.1.

5.4 Správa úloh

Pro zajištění funkcí spojených se správou úloh byly v aplikaci implementovány dva základní kontejnery `ProjectTasksTabContainer` a `TaskFormContainer`. Obdobně jako v předchozích částech je i zde první z kontejnerů tvořen tabulkou, která v tomto případě slouží pro zobrazení přehledu všech úloh definovaných v rámci projektu. Součástí tohoto přehledu je kromě názvu úlohy i její stav a několik ovládacích prvků s nimiž souvisí jedny z nejdůležitějších funkcí kontejneru. K nim patří například spouštění a zastavování úloh, které podporuje i autentifikaci uživatele v cílovém prostředí prostřednictvím privátního SSH klíče a uživatelského jména. Pro tyto účely nabízí kontejner dialogové okno představované komponentou `TaskCredentialModal`, jehož součástí je odpovídající



Obrázek 5.2: Správa úloh

formulář. Zadané údaje jsou pak připojeny k požadavku na změnu stavu úlohy, který je odeslán na řídící server pomocí modulu `TaskApi`.

U běžících úloh je dále dostupná i funkce spojená s odesláním požadavku na vyhodnocení vstupních dat. Pro jeho vytvoření je připravena komponenta `TaskFileTableModal`, která umožňuje z projektových souborů vybrat soubor jenž bude představovat vstupní data. Po jeho odeslání je získán identifikátor požadavku, který je následně využit pro opakované dotazování na přítomnost výstupního souboru. V případě jeho nalezení je automaticky spuštěno stahování.

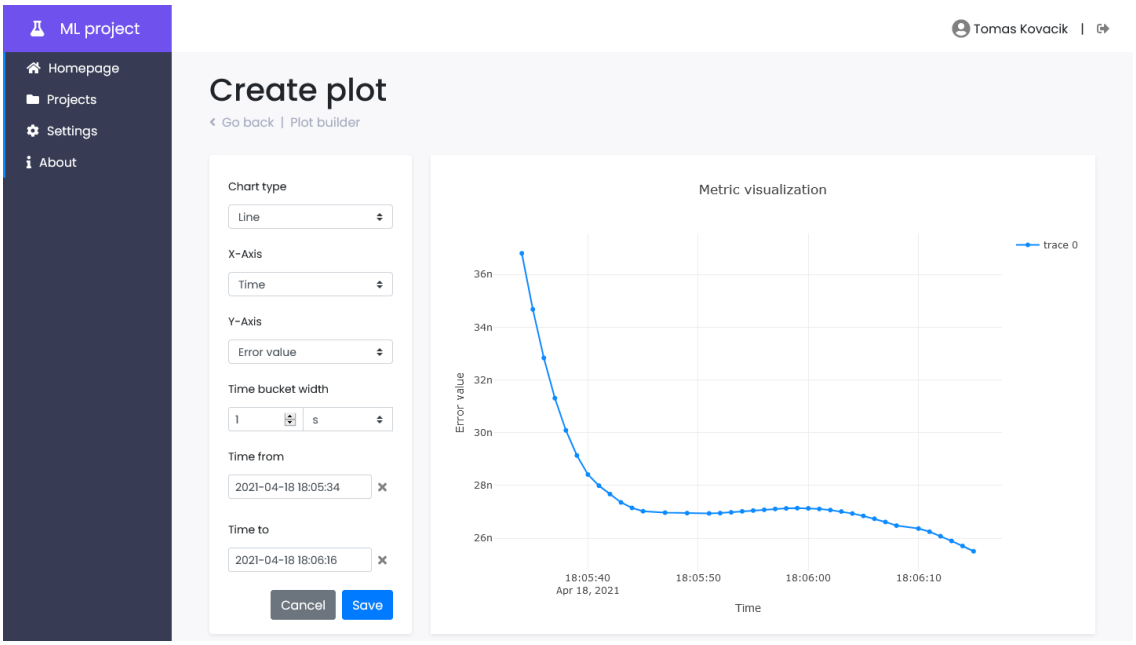
Celkovou podobu stránky určenou pro správu úloh je možné vidět na obrázku 5.2.

S úlohami jsou dále spojeny funkce týkající se jejich vytváření a editace, o které se stará druhý z kontejnerů s názvem `TaskFormContainer`. Jeho součástí je dynamický formulář, který umožňuje definovat základní vlastnosti pro všechny dostupné typy cílových prostředí. Kromě těchto vlastností lze v aplikaci pomocí dalších kontejnerů specifikovat i seznam souborů potřebných pro běh nebo seznam parametrů, které mají být úloze předány při spuštění.

5.5 Vytváření vizualizací

Grafická vizualizace představuje jeden ze způsobů umožňující interpretaci metrik získaných v průběhu učení neuronové sítě. Pro vytváření takových vizualizací byl v aplikaci implementován kontejner `MetricPlotBuilderContainer`, jehož základem jsou dva panely.

První z panelů obsahuje dynamický formulář, pomocí něhož mohou uživatelé definovat parametry určené pro sestavení jednoho ze tří grafů mezi něž patří graf spojnicový, sloupcový a graf



Obrázek 5.3: Tvorba vizualizací

histogramu. V případě spojnicového a sloupcového grafu tyto parametry zahrnují seskupující atribut, typ metriky a několik specifických parametrů. Naopak u histogramu lze zvolit pouze metriku a pro ni dva parametry reprezentující rozsah a počet skupin. Ve spojitosti s tímto dělením byla nakonec rozdělena i implementace samotného formuláře, a to do několika částí představující jednotlivé logické celky.

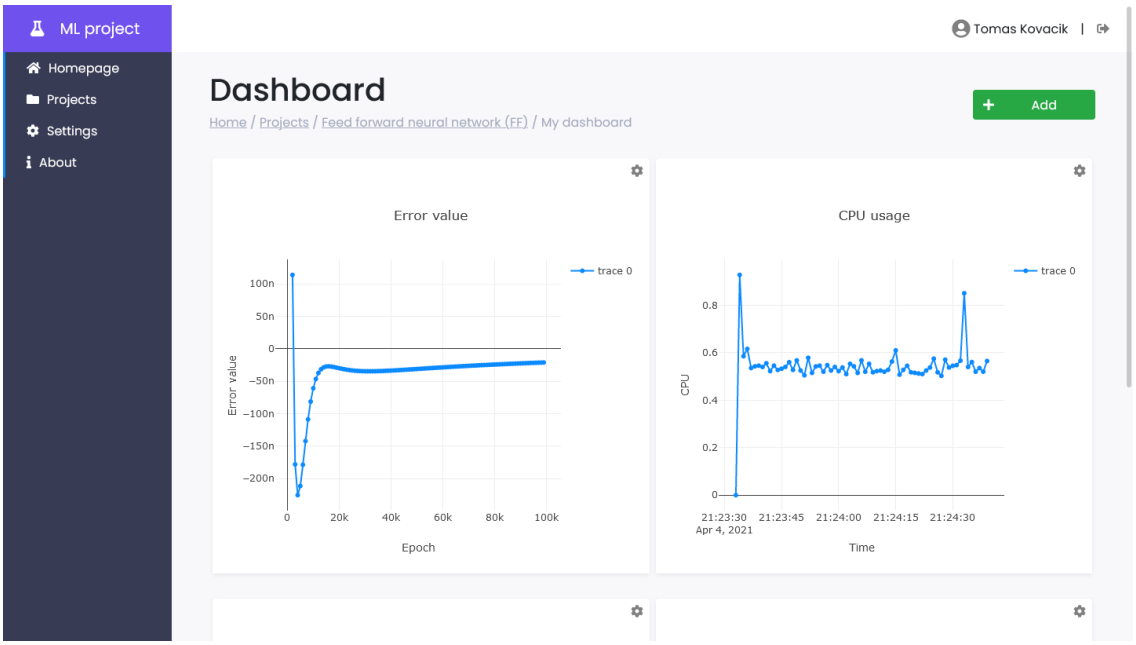
Druhý z panelů následně obsahuje vizualizaci výsledného grafu, která je zajištěna kontejnerem `MetricPlotContainer`. V něm se kromě využití knihovny Plotly sloužící k vykreslení grafu nachází i potřebná logika související s načtením a zpracováním dat z řídícího serveru. Pro účely komunikace s řídícím serverem je zde využit speciální objekt sestavený z parametrů předaných samotnému kontejneru, který definuje, jaké data mají být získány.

Náhled na celkovou podobu stránky s příslušným kontejnerem je možný vidět na obrázku 5.3.

5.6 Vytváření a úprava skupin vizualizací

Pro vytváření a úpravu skupin vizualizací byl v aplikaci implementován kontejner `DashboardContainer` jehož hlavní část je představována mřížkou. Do ní mohou uživatelé ve formě karet umístit některou z předem připravených vizualizací dostupných v rámci projektu, kterou je možné vyhledat a vybrat pomocí komponenty dialogového okna `VisualizationSearchModal`.

Každá takto přidaná karta s vizualizací pak nabízí několik možností úprav zahrnujících jejich přemístění na jinou pozici nebo odstranění z dané skupiny. V případě zvolení možnosti pro změnu pozice je uživatel pomocí dialogového okna vyzván k její volbě z poskytnutého seznamu. Po potvrzení



Obrázek 5.4: Náhled na skupinu vizualizací

změny je požadavek odeslán na řídící server, který kromě přemístění karty na novou pozici provede i reorganizaci ostatních karet tak, aby nevznikla žádná mezera. Obdobný postup je dále uplatněn i u odstraňování karet, avšak na rozdíl od editace pozice je zde zobrazen pouze dialog pro potvrzení operace.

Obsah jednotlivých karet je dále tvořen komponentami typu `PlotContainer`, které jako své nastavení přejímají objekt typu `MetricQuery`. Tento objekt je v kontejneru `DashboardContainer` získáván z řídícího serveru při své inicializaci a při každé provedené změně ve skupině vizualizací. O načtení potřebných dat se tak v tomto případě stará každá z karet sama nezávisle na ostatních.

Ukázku stránky umožňující správu skupiny vizualizací je možné vidět na obrázku 5.4.

5.7 Správa uživatelů

Správa uživatelů je v aplikaci řešena pomocí kontejneru `UserTabContainer`, který tvoří samostatnou část konfigurace dostupnou skrze hlavní menu. Přístup do této sekce je omezen pouze na administrátorské účty a není proto viditelná pro běžné uživatele.

Samotný kontejner je tvořen několika ovládacími prvky a tabulkou nesoucí seznam všech uživatelských účtů. Základní ovládací prvek v podobě tlačítka je umístěn v pravém horním rohu a nabízí možnost přidání nového uživatele skrze kontejner `UserFormContainer`. Ten je tvořen samostatnou stránkou s formulářem, který spolu s vytvářením nových uživatelských účtů nabízí i možnost jejich editace. Pro tyto účely nabízí formulář několik vstupních polí, mezi nimiž nechybí ani textové pole pro nastavení hesla nebo volby přihlašovacího jména.

Ve formuláři nicméně není možné pro uživatele definovat jejich roli, a proto je pro tento účel v tabulce uživatelů vytvořen sloupec s možností její změny. Provedené úpravy skrze tabulku jsou pak aplikovány okamžitě a není nutné je ukládat zvlášť.

Kromě možnosti editace se na stránce s výpisem uživatelů nachází i možnost jejich smazání, která vyžaduje potvrzení skrze dialogové okno. V případě potvrzení změn je následně odeslán požadavek na řídící server spolu se zavoláním akce pro aktualizaci tabulky.

Kapitola 6

Nasazení a testování

Tato kapitola se zabývá nasazením výsledného systému do produkčního prostředí a jeho testováním s využitím vzdáleného výpočetního serveru a superpočítače Barbora.

6.1 Příprava prostředí a nasazení

Pro nasazení systému bylo zvoleno produkční prostředí využívající operační systém Ubuntu. V něm byla provedena řada úprav složená z několika kroků, které měly za cíl jej připravit pro provoz trojice aplikací zahrnujících řídící server, autentifikační server a webový server určený pro provoz části frontend. Přehled všech provedených kroků je uveden níže:

1. Instalace softwaru
2. Vytvoření adresářové struktury
3. Sestavení projektu a nahrání aplikací na server
4. Konfigurace aplikací řídícího a autentifikačního serveru
5. Vytvoření systemd jednotek a uživatelského účtu pro provoz aplikace prostřednictvím služeb
6. Spuštění systému a kontrola jeho funkčnosti

Jedním z prvních kroků přípravy byla instalace a konfigurace nezbytného softwarového vybavení, které bylo v tomto případě složené z běhového prostředí Java 11, databázového serveru TimescaleDB a webového serveru nginx. Během tohoto kroku byly vytvořeny dvě databázová schémata pro obě ze serverových částí systému a do aplikace nginx byla přidána konfigurace pro realizaci reverzní proxy a webového serveru.

Po dokončení instalace a konfigurace byla následně vytvořena základní adresářová struktura do níž byly později umístěny sestavené aplikace řídícího a autentifikačního serveru spolu s jejich

konfigurací. Celá část řídícího serveru byla v tomto případě umístěna do složky */opt/ml-project*, ve které byly vytvořeny další podadresáře pro jednotlivé moduly. Část aplikace běžící ve webovém prohlížeči pak byla umístěna do příslušné složky v adresáři */var/www*.

V dalším kroku následovala konfigurace obou modulů, která zahrnovala nastavení několika položek týkajících se připojení k jiným systémům a komponent pro spouštění a řízení klientských aplikací ve vybraných prostředích. Mezi nejdůležitější z těchto položek patřily zejména adresy a přihlašovací údaje určené pro připojení k databázovému serveru, adresy pro připojení ke vzdálenému výpočetnímu serveru a superpočítači a v neposlední řadě také samotná adresa řídícího serveru. Kromě toho zde bylo nutné specifikovat i cesty k pracovním složkám pro jednotlivá prostředí nebo cestu k souboru *known_hosts* do nějž byly později manuálně vloženy potřebné záznamy pro připojení k oběma vzdáleným prostředím.

V předposledním kroku již jen následovalo vytvoření *systemd* jednotek pro jednotlivé moduly, které umožňovaly jejich běh v rámci systémových služeb. Díky tomu mohly být obě části řízeny a sledovány běžnými nástroji zprostředkovanými operačním systémem. Pro zajištění jejich bezpečného běhu byl v rámci tohoto kroku vytvořen i uživatelský účet s minimálními oprávněními, který nedisponoval privilegovaným přístupem.

Po dokončení všech příprav byl celý systém spuštěn a byla ověřena jeho funkčnost a spolupráce s nakonfigurovaným prostředím vzdáleného výpočetního serveru a superpočítače.

6.2 Testování

V průběhu vývoje bylo průběžně prováděno testování zejména manuální formou v lokálním prostředí, a to především kvůli povaze celého systému, jehož části věnující se spouštění a řízení výpočetních úloh jsou uzpůsobeny pro využití se vzdáleným výpočetním serverem a superpočítačem.

Pro tento případ byla vytvořena vzorová implementace realizující učení neuronové sítě pomocí algoritmu Backpropagation, která využívala nejrůznější funkce klientské knihovny. Mezi nimi byly například funkce pro záznam metrik, sledování změn v parametrech nebo funkce pro zpracování požadavků na vyhodnocení vstupních dat. Výsledná implementace byla současně umístěna do balíčku `com.vsb.mlprojectmanager.client.example`, který je součástí projektu klientské knihovny.

S využitím této ukázkové implementace bylo následně provedeno několik testů ověřujících zmíněné funkce ale i samotné spouštění a zastavování úloh v jednotlivých prostředích. Zde byly zvláště v případě prostředí superpočítače otestovány různé konfigurace zahrnující například rozdílný počet alokovaných uzlů nebo špatné nastavení jejich počtu jader. Kromě toho bylo také otestováno nahrávání výstupních souborů vygenerovaných úlohou za jejího běhu nebo správné ukončení úlohy v případě nějaké chyby.

Dále byl otestován i systém nasazený do produkčního prostředí, kde byla provedena stejná sada testů. Při nich byla odhalena nefunkčnost spojení klientské aplikace s řídícím serverem pomocí

technologie SSE, kterou se bohužel nepodařilo v rámci práce vyřešit. Její přítomnost je podle dosavadních zjištění způsobena nejspíše chybějící nebo nevhodnou konfigurací webového serveru nginx.

Kapitola 7

Závěr

V rámci této diplomové práce byl vyvinut systém pro monitorování výpočtů realizujících učení neuronových sítí a jejich spouštění ve formě úloh na vzdáleném výpočetním serveru nebo superpočítači.

Výsledný systém poskytuje přehledné grafické uživatelské rozhraní, které umožňuje vytváření definic úloh, jejich spouštění a řízení, správu souborů a datových sad nebo úpravu nejrůznějších parametrů ovlivňujících učení neuronových sítí. Samotné monitorování je pak realizováno pomocí knihovny, která nabízí aplikační rozhraní umožňující klientským aplikacím využívat kromě běžných funkcí pro záznam metrik i řadu rozšiřujících funkcí. Ty zahrnují například funkce spojené se sledováním parametrů úlohy nebo s nahráváním souborů na řídící server.

Pro zpracování naměřených dat dále aplikace nabízí možnost i jejich grafické vizualizace pomocí několika druhů grafů. Ty lze následně využít pro sestavení skupin vizualizací díky nimž mohou uživatelé u každé z úloh výsledky přehledně sledovat nebo je porovnávat napříč několika úlohami v rámci jednoho projektu.

V průběhu vývoje se počítalo také s pozdějším rozšířením systému, do něhož by mohly být v budoucnu přidány například nové typy grafických vizualizací nebo by celý systém mohl být rozšířen o možnost zachycení standardního a chybového výstupu klientské aplikace generovaného za jejího běhu. Pomocí těchto informací by následně uživatelé mohli získat lepší přehled o chování své aplikace a mohli by tak snadněji opravit případné chyby, které za jejího běhu nastanou. Kromě těchto rozšíření by v aplikaci mohl být nahrazen i modul autentifikačního serveru u něhož se v poslední fázi vývoje po aktualizaci jeho hlavních závislostí zjistilo, že již nejsou podporované a byly označeny za zastaralé. Vhodnou náhradou by tak v tomto případě mohl být univerzitní systém pro přihlašování nebo jiné řešení.

Literatura

1. SSH Protocol – Secure Remote Login and File Transfer. *SSH.com* [online]. Waltham: SSH Communications Security, c2020 [cit. 2021-03-09]. Dostupné z: `https://www.ssh.com/ssh/protocol/`.
2. Configure SSH key based secure authentication. *SSH.com* [online]. Waltham: SSH Communications Security, c2020 [cit. 2021-03-09]. Dostupné z: `https://www.ssh.com/ssh/key/`.
3. Passphrase - What it is, how to use. *SSH.com* [online]. Waltham: SSH Communications Security, c2020 [cit. 2021-03-09]. Dostupné z: `https://www.ssh.com/ssh/passphrase`.
4. SSH Command - Usage, Options, Configuration. *SSH.com* [online]. Waltham: SSH Communications Security, c2020 [cit. 2021-03-09]. Dostupné z: `https://www.ssh.com/ssh/command/`.
5. Resources Allocation Policy. *Home - IT4Innovations Documentation* [online]. Ostrava: IT4Innovations, c2013-2021 [cit. 2021-03-09]. Dostupné z: `https://docs.it4i.cz/general/resources-allocation-policy/`.
6. Job Submission and Execution. *Home - IT4Innovations Documentation* [online]. Ostrava: IT4Innovations, c2013-2021 [cit. 2021-03-09]. Dostupné z: `https://docs.it4i.cz/general/job-submission-and-execution/`.
7. Storage. *Home - IT4Innovations Documentation* [online]. Ostrava: IT4Innovations, c2013-2021 [cit. 2021-03-09]. Dostupné z: `https://docs.it4i.cz/barbora/storage/`.
8. Data Model. *TimescaleDB Documentation* [online]. New York: Timescale, c2021 [cit. 2021-03-09]. Dostupné z: `https://docs.timescale.com/latest/introduction/data-model`.
9. TimescaleDB Overview. *TimescaleDB Documentation* [online]. New York: Timescale, c2021 [cit. 2021-03-09]. Dostupné z: `https://docs.timescale.com/latest/introduction`.
10. Architecture & Concepts. *TimescaleDB Documentation* [online]. New York: Timescale, c2021 [cit. 2021-03-09]. Dostupné z: `https://docs.timescale.com/latest/introduction/architecture`.
11. Why Use TimescaleDB over Relational DBs? *TimescaleDB Documentation* [online]. New York: Timescale, c2021 [cit. 2021-03-06]. Dostupné z: `https://docs.timescale.com/latest/introduction/timescaledb-vs-postgres`.

12. TimescaleDB API Reference. *TimescaleDB Documentation* [online]. New York: Timescale, c2021 [cit. 2021-03-09]. Dostupné z: `https://docs.timescale.com/latest/api`.

13. InfluxDB. *InfluxDB: Purpose-Built Open Source Time Series Database | InfluxData* [online]. San Francisco: InfluxData, c2021 [cit. 2021-03-07]. Dostupné z: `https://www.influxdata.com/products/influxdb/`.

14. TimescaleDB vs. InfluxDB: Purpose built differently for time-series data. *Timescale Blog* [online]. New York: Timescale, c2021 [cit. 2021-03-09]. Dostupné z: `https://blog.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/`.

15. InfluxQL functions. *InfluxData Documentation* [online]. San Francisco: InfluxData, c2021 [cit. 2021-03-09]. Dostupné z: `https://docs.influxdata.com/influxdb/v1.8/query_language/functions/`.

16. Spring Boot. *Spring | Home* [online]. Palo Alto: VMware, c2021 [cit. 2021-03-28]. Dostupné z: `https://spring.io/projects/spring-boot`.

17. Hibernate ORM. *Hibernate. Everything data. - Hibernate* [online]. Raleigh: Red Hat [cit. 2021-03-28]. Dostupné z: `https://hibernate.org/orm/`.

18. Great Reasons for Using jOOQ. *JOOQ: The easiest way to write SQL in Java* [online]. St.Gallen: Data Geekery, c2009-2021 [cit. 2021-03-28]. Dostupné z: `https://www.jooq.org/`.

19. Apache MINA SSHD. *GitHub: Where the world builds software · GitHub* [online]. San Francisco: GitHub, c2021 [cit. 2021-03-28]. Dostupné z: `https://github.com/apache/mina-sshd`.

20. GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. *Návrh programů pomocí vzorů: stavební kameny objektově orientovaných programů*. Praha: Grada Publishing, 2003. Moderní programování. ISBN 80-247-0302-5.

21. DARWIN, Ian F. *Java Cookbook*. 2nd ed. Sebastopol: O'Reilly, 2004. ISBN 0-596-00701-9.

22. LYČKA, Patrik. *Modul RBM a DBM pro program Modeler neuronových sítí* [online]. Ostrava, 2020 [cit. 2021-04-10]. Dostupné z: `http://hdl.handle.net/10084/140538`. Diplomová práce. Vysoká škola báňská - Technická univerzita Ostrava.

23. SCHILDT, Herbert. *Mistrovství - Java*. Brno: Computer Press, 2014. ISBN 978-80-251-4145-8.

24. React. *React – A JavaScript library for building user interfaces* [online]. Menlo Park: Facebook, c2021 [cit. 2021-04-01]. Dostupné z: `https://reactjs.org`.

25. Redux. *Redux - A predictable state container for JavaScript apps. | Redux* [online]. Dan Abramov, c2015-2021 [cit. 2021-04-01]. Dostupné z: `https://redux.js.org`.

26. Redux-Saga. *Redux-Saga - An intuitive Redux side effect manager. | Redux-Saga* [online]. Redux-Saga, c2021 [cit. 2021-04-01]. Dostupné z: `https://redux-saga.js.org`.

27. Reactstrap. *Reactstrap - React Bootstrap 4 components* [online] [cit. 2021-04-01]. Dostupné z: `https://reactstrap.github.io`.

28. React Router. *React Router: Declarative Routing for React.js* [online]. React Training, c2021 [cit. 2021-04-01]. Dostupné z: `https://reactrouter.com`.

Příloha A

Obsah elektronických příloh

ml-project-manager-backend - složka obsahující zdrojové kódy části backend. Obsahuje moduly autentifikačního a řídícího serveru.

ml-project-manager-frontend - složka obsahující zdrojové kódy části frontend.

ml-project-manager-client - složka obsahující zdrojové kódy klientské knihovny. Součástí zdrojových kódů je i ukázkový příklad nacházející se v balíčku `com.vsb.mlprojectmanager.client.example`.

Návod ke spuštění.txt - textový soubor s návodem pro sestavení a spuštění jednotlivých částí.